TECHNISCHE UNIVERSITÄT
ILMENAU

Entwicklung eines Modells zur Anwendung inferenzfähiger Ontologien im Software Engineering

DISSERTATION

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau
von M.Sc. Franz Felix Füßl

Gutachter:
1. Dr.-Ing Detlef Streitferdt - TU Ilmenau
2. apl. Prof. Dr.-Ing. habil. Rainer Knauf - TU Ilmenau
3. Prof. Dr.-Ing. habil. Josef Börcsök - Universität Kassel

Tag der Einreichung: 01.07.2016
Tag der wissenschaftlichen Aussprache: 21.12.2016

urn:nbn:de:gbv:ilm1-2016000726

Danksagung

Maßgebend für den Erfolg meiner Arbeit war eine vielseitige fachliche und menschliche Unterstützung. Dafür möchte ich herzlich danken.

Mein besonderer Dank gilt Professor Detlef Streitferdt, der mir während der vergangenen drei Jahre jederzeit mit Rat und Tat zur Seite stand. Ohne seine Unterstützung wäre eine Arbeit an diesem interessanten Forschungsthema nicht möglich gewesen. Auch Professor Rainer Knauf möchte ich meinen Dank ausprechen. Durch sein außerordentliches Interesse an der Thematik und seine Expertise konnte ich persönliche Erkenntnisse auf dem Gebiet der künstlichen Intelligenz in die Arbeit einfließen.

Während meiner Zusammenarbeit mit dem Fachgebiet Softwarearchitekturen und Produktlinien habe ich wichtige fachliche Erfahrungen in Wissenschaft und Forschung sammeln können. Daher gilt mein Dank allen Kollegen und Kolleginnen, die mich auf meinem Weg begleitet haben.

Zu guter Letzt danke ich meiner Frau Anne und meiner Familie für die unendliche Geduld, das entgegengebrachte Interesse und die vielen kleinen Hilfeleistungen.

Kurzfassung

Die Arbeit beschreibt die Entwicklung und Anwendung eines Modells zur Repräsentation und Verarbeitung von Wissen auf dem Gebiet des Software Engineering. Im Fokus stehen die Strukturierung, Visualisierung und automatisierte Weiterverarbeitung von Expertenwissen unter Einbezug modellbasierter Inferenzmechanismen. Dabei dienen verschiedene Deduktionsalgorithmen zur Ableitung automatisierter Schlussfolgerungen, die zur Lösung einer Software Engineering Fragestellung beitragen. Darüber hinaus werden unterschiedliche Möglichkeiten dargestellt, um Wissen induktiv, automatisiert aus bereits modelliertem Wissen ableiten zu können. Das entwickelte Modell soll Akteuren des Software- und Knowledge Engineering gleichermaßen dazu dienen, Expertenwissen in Entscheidungsprozesse einzubeziehen, um somit die Entwicklung von Softwarelösungen zu beschleunigen.

„Intelligence is mind implemented by any patternable kind of matter."

Allen Newell, Herbert Simon

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Umfangreiche Softwareentwicklungsprojekte, wie die Anfertigung einer Softwareproduktlinie oder die Erstellung komplexer und sicherheitsrelevanter Anwendungssysteme, bestehen aus den Teilaufgaben Anforderungsmanagement, Softwarearchitektur, Implementierung, Test und Integration. Nach Abschluss des Projektes liegen die Ergebnisse der einzelnen Teilaufgaben in Form verschiedener Projektartefakte, wie Anforderungs- und Architekturdokumente, Quellcode oder Testergebnisse vor. Für einen geordneten Ablauf des Projektes muss der gesamte Entstehungsprozess dieser Projektartefakte vom Projektmanagement koordiniert und kontrolliert werden. Das Endergebnis des Softwareentwicklungsprojektes bildet bei erfolgreicher Projektumsetzung eine Software, die allen zuvor definierten Anforderungen und Projektzielen gerecht wird. Ein Projekterfolg ist demnach nur schwer vorhersehbar, da erst nach Fertigstellung der Software und nach Durchführung aller erforderlichen Tests festgestellt werden kann, inwieweit die Projektumsetzung der Planung entsprach und inwiefern das Resultat die Bedürfnisse des Auftraggebers widerspiegelt. Besonders bei komplexen Softwareentwicklungsvorhaben ist eine vollständige Planung jedoch schwer durchführbar, da das Projekt von verschiedenen Rahmenbedingungen und unvorhersehbaren Ereignissen beeinflusst werden kann. Zudem sind diese Einflussfaktoren von Projekt zu Projekt verschieden, was eine Abschätzung der Risikofaktoren erschwert.

Um den definierten Projekterfolg erzielen zu können, muss das Softwareprojekt so gestaltet werden, dass die Projektumsetzung entsprechend der Planung verläuft. Dazu ist die Integration aller Einflüsse und Rahmenbedingungen erforderlich, die sich negativ auf den Projekterfolg auswirken könnten. Beispielsweise muss zu Beginn des Projektes die Auswahl softwaregestützter Werkzeuge, die zur Projektumsetzung dienen, den Erfahrungen der Projektbeteiligten und den Anforderungen an das Projektergebnis entsprechen. Soziale Faktoren, wie Motivation und Kenntnisse des Einzelnen und des Teams oder Disziplin und

Expertise der beteiligten Personen werden dabei meist unzureichend berücksichtigt oder sind zu Projektbeginn gänzlich unbekannt. Auch technische Rahmenbedingungen, wie Programmiersprache, verwendetes Betriebssystem oder Hardwarebeschränkungen können die Auswahl geeigneter Hilfsmittel zur Projektumsetzung und damit den Projektverlauf beeinflussen.

Alle Faktoren zu berücksichtigen und ein Projekt entsprechend seiner Einflüsse zu gestalten ist eine komplexe Aufgabe, die reichlich Erfahrung und die Fähigkeit voraussetzt, vorausschauend über längere Zeiträume zu planen. Um Verantwortlichen diese Aufgabe zu erleichtern, ist ein eigens zu diesem Zweck entwickeltes Modell sinnvoll, das unter Berücksichtigung der erwähnten Faktoren eine Zusammensetzung der Projekthilfsmittel vorschlägt und unter Umständen sogar die Auswahl verschiedener Werkzeuge und Vorgehensweisen selbständig vornimmt. Dieses Modell soll dazu dienen, Knowledge Engineering auf dem Gebiet der Softwareentwicklung zu unterstützen, indem die Teilaufgaben „Akquirieren“, „Strukturieren“ und „Visualisieren“, speziell auf Wissen über Softwareentwicklungsprojekte (z.B. soziale und technische Einflüsse, softwaregestützte Werkzeuge oder Softwareentwicklungsmethoden) angewandt wird.

1.2 Zielsetzung

Ziel dieser Dissertation ist die Erarbeitung eines Modells, um Wissen auf dem Gebiet der Softwareentwicklung gezielt in neue Entwicklungsvorhaben einzubeziehen und diese an projektabhängige Störgrößen und Rahmenbedingungen anpassen zu können. Die erforderlichen Teilaufgaben der Softwareentwicklung sollen durch Wissen, welches mithilfe des Modells digital überführt wird, projektspezifisch unterstützt werden, beispielsweise die Auswahl von Softwarewerkzeugen, die Prüfung einer geeigneten Softwareentwicklungsmethode oder die Zusammenstellung eines Teams entsprechend ihrer Kenntnisse und Fähigkeiten.

Das Modell soll insbesondere folgenden zum Teil ambivalenten Kernanforderungen gerecht werden:

1. Das im Modell enthaltene Wissen soll grafisch und mathematisch darstellbar sein, um einen hohen Grad an Formalisierung zu erreichen.

2. Es existieren Vorgaben zu verschiedenen Assoziationstypen, die zur Verknüpfung von Wissenselementen dienen. Eigene Assoziationstypen des Modellierers sind nicht möglich, um den Formalisierungsgrad zu stabilisieren.
3. Ein weiteres Formalisierungsmerkmal sind vorgegebene Elementklassen, die dem Modellierer beim Überführen des Wissens zur Verfügung stehen, wie die Unterteilung in Konkretisierung und Abstraktion oder die Integration von Handlungen, Aufgaben oder Tätigkeiten.
4. Es wird eine Möglichkeit dargeboten, hierarchisch vorliegende Wissenskonstrukte zu zusammenhängendem Wissen zu kombinieren.
5. Das Modell bietet die Möglichkeit, automatisiert aus dem modellierten Wissen Ableitungen, Entscheidungen und andere Schlussfolgerungen zu treffen, zum Beispiel durch Induktions- und Deduktionsalgorithmen.
6. Das Modell liefert eine Möglichkeit, Datenerfassung bzw. Nutzereingaben zu modellieren und in Schlussfolgerungsprozesse zu integrieren.

Das Zielvorhaben soll zur IT-gestützten Entscheidungsfindung dienen, diese in geeigneter Weise kommunizieren und dabei alle notwendigen Schlussfolgerungsmechanismen enthalten, um Erfolgs- und Risikofaktoren möglichst frühzeitig in die Planung eines Softwareprojekts zu integrieren.

1.3 Methodik

Das Dissertationsziel soll in drei Schritten realisiert werden. Zunächst erfolgt die Beschreibung aller erforderlichen Grundlagen, einschließlich einer Begriffsbestimmung des Terminus „Knowledge Engineering“. Des Weiteren werden fachverwandte Themen, wie Knowledge Modeling und Ontology Engineering voneinander abgegrenzt. Letztere ist für die Erarbeitung der Zielstellung von besonderer Bedeutung und erhält daher eine ausführliche Betrachtung der wesentlichen Teilgebiete. Die Darstellung des aktuellen Forschungsstandes soll durch eine Literaturrecherche nach WEBSTER UND WATSON (2002) eingeleitet werden, bei der Arbeiten mit ähnlichen Zielstellungen analysiert werden.

Aus den Ergebnissen der Literaturanalyse erfolgt im zweiten Schritt die mathematische, visuelle und textuelle Beschreibung des Modells. Der Fokus hierbei liegt auf der Integration eines besonders hohen Formalisierungsgrades, um die eingangs beschriebene Digitalisierung von Expertenwissen im Bereich der Softwareentwicklung standardisiert zu gestal-

ten und darüber hinaus eine Grundlage zur Ableitung von Schlussfolgerungen zu schaffen. Die Schlussfolgerungen basieren auf dem im Modell enthaltenem Wissen und der zugrundeliegenden Struktur. Diese werden bei der Erarbeitung des Modells als Deduktionsalgorithmen bezeichnet und sollen maßgeblich zur Entscheidungsfindung unter Berücksichtigung von Eingabeparametern dienen.

Zur Ermittlung der Anwendbarkeit und zur Evaluierung des Systems wird ein Prototyp entwickelt, der als Webanwendung umgesetzt wird und das erarbeitete Modell als Architekturgrundlage nutzt. Das im Prototyp verarbeitete Wissen orientiert sich dabei an Experten des Software Engineering. Dieses Wissen wird für Entscheidungsprozesse herangezogen, die anschließend durch eine Evaluierung auf einen erfolgreichen Einsatz des Modells in Produktivumgebungen geprüft werden sollen.

1.4 Aufbau

Die Dissertationsschrift ist in sechs Abschnitte gegliedert. Nach der Einleitung folgen in Kapitel 2 Begriffsbestimmungen sowie die Beschreibung des aktuellen Stands der Forschung. Im Wesentlichen werden die zentralen Bereiche des Knowledge Engineering und derzeitige Methoden zur Repräsentation von Wissen betrachtet. Hierbei liegt der Fokus auf dem Ontology Engineering. Auf Basis dieser Grundlagen wird in Abschnitt 2.1 eine Literaturanalyse nach WEBSTER UND WATSON (2002) durchgeführt, womit die wichtigsten bis dato veröffentlichten Schriften zusammengetragen und ihre Bedeutung für die Bearbeitung der vorliegenden Zielstellung aufgeführt wird. Die mathematische und textuelle Beschreibung des Modells folgt in Kapitel 3. Dabei werden die Modellierungstechnik aufgezeigt und verschiedene Möglichkeiten beschrieben, Wissen auf geeignete Weise durch das Modell zu digitalisieren.

Neben der Modellierung von Wissen soll das zu entwickelnde System dazu im Stande sein, auf Basis verschiedener Methoden zu Lernen und Gelerntes anzuwenden. Daher soll die digitale Repräsentation von Wissen genutzt werden, um in Kapitel 1 Möglichkeiten zur Inferenzintegration darzulegen. Deren Ergebnisse können als Lösung einer softwarespezifischen Fragestellung oder als Mittel zur Wissensableitung im Allgemeinen betrachtet werden. Darüber hinaus soll Kapitel 1 dazu dienen, einfache Zusammenhänge von Informationen und Abhängigkeitsstrukturen unterschiedlicher Wissensfragmente zu finden und lesbar aufzubereiten. Abschnitt 4.2 beschreibt, wie Lerncharakteristika in das Modell inte-

griert werden. Es untersucht dabei Möglichkeiten, aus einfachen semantischen und pragmatischen Zusammenhängen neues Wissen zu konstruieren, sowie mithilfe induktiver Inferenz Schlussfolgerungen zu bereits vorhandenem Wissen zu ziehen.

Ein genaue Darstellung der Evaluierungsziele und –vorgehensweise erfolgt in Kapitel 1. Hierbei wird zunächst der softwarebasierte Modellprototyp einschließlich aller an ihn gestellten Anforderungen betrachtet. Anschließend erfolgen eine Beschreibung der Architektur und ausgewählte Beispiele der Implementierung. Mithilfe dieses Prototyps wird eine Evaluierung der Anwendbarkeit des Gesamtvorhabens durchgeführt. Die genauen Erläuterungen der Herangehensweise und die Analyse der Ergebnisse erfolgen in den Abschnitten 5.2 und 5.3.

2 Stand der Technik

2.1 Vorgehensweise zur Literaturanalyse

Zur Identifikation relevanter Literatur wird eine Analyse nach WEBSTER UND WATSON (2002) durchgeführt. Ihre Methode wird häufig als Einstieg in ein Forschungsvorhaben oder als Ansatz zur Analyse des derzeitigen Stands der Technik, insbesondere im Bereich der Informationsverarbeitung, verstanden. Durch die Methode von WEBSTER UND WATSON (2002) kann unter Verweis auf bestehende Publikationen eine gegebene Thematik genauer spezifiziert oder erweitert werden. Darüber hinaus kann die Methode zur Entwicklung neuer Konzepte oder Modelle dienen, bei denen theoretische Ansätze vorangegangener Arbeiten die Basis bilden.

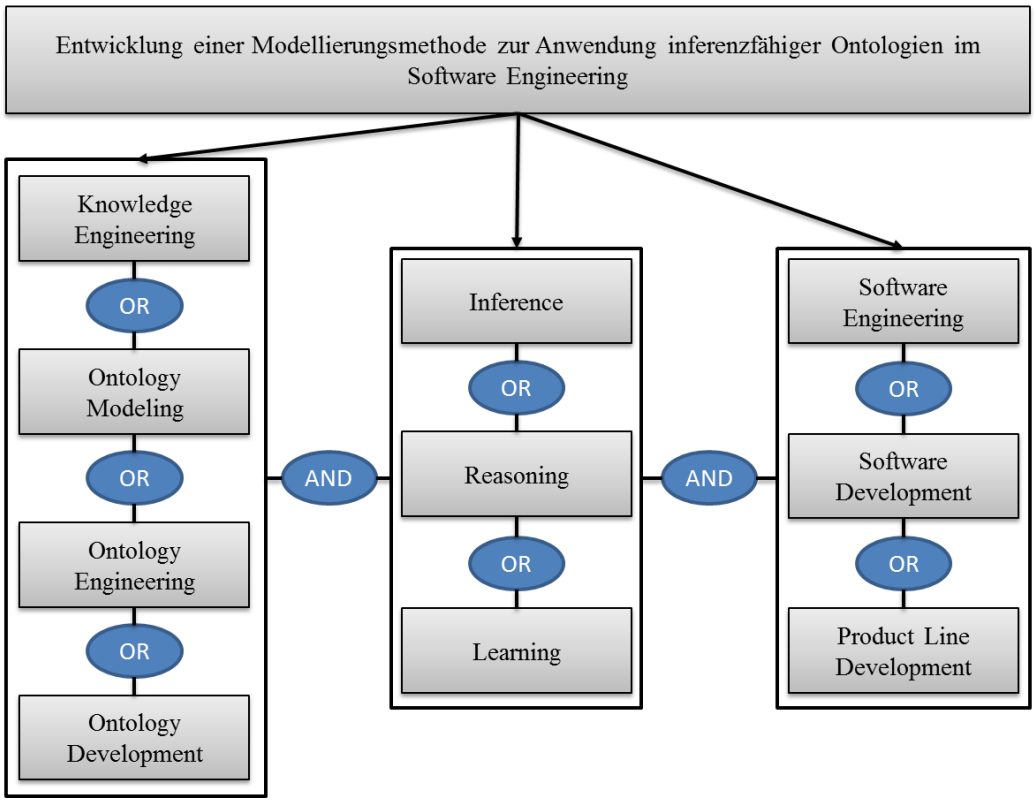


Abb. 1 Basis der Literaturanalyse

Die Suchbegriffe der Literaturanalyse basieren die Thematik des Forschungsprojektes. Im Rahmen der Untersuchung derzeitiger Modellierungsmethoden, die zum Ontology Engineering in der Softwareentwicklung dienen, bilden die in Abb. 1 dargestellten Schlagwörter die Basis der Literaturanalyse.

Die wissenschaftlichen Publikationsdatenbanken Elsevier Science Direct, IEEE Xplore Digital Library und Springer Link werden zur Identifikation der relevanten Literatur genutzt. Wie Abb. 1 zeigt, ergeben sich 36 Suchausdrücke, die zur gezielten Suche in den jeweiligen Suchmaschinen bzw. Publikationsbibliotheken dienen. Anschließend können anhand der ermittelten Artikel und Autoren durch Rückwärts- und Vorwärtssuche weiteres Literaturmaterial eruiert werden. Der Veröffentlichungszeitraum wird zwischen 2000 und 2016 festgesetzt.

Bei den zu untersuchenden Literaturquellen werden insbesondere die in der Zielstellung dieser Arbeit aufgeführten Anforderungen an die enthaltenen Lösungsansätze gestellt:

1. Die Autoren beschreiben eine Methode zum Modellieren von Wissen (mindestens aus der Domäne des Software Engineering) und beziehen sich dabei auf grafische und mathematische Darstellungen.
2. Der untersuchte Ansatz bietet einen hohen Grad an Formalisierung. Er enthält Vorgaben zu verschiedenen Assoziationstypen (wie partOf, has, etc.), die zur Verknüpfung von Wissenselementen dienen. Eigene Assoziationstypen des Modellierers sind nicht möglich. Weiteres Formalisierungsmerkmal sind vorgegebene Elementklassen, die dem Modellierer beim Überführen des Wissens zur Verfügung stehen, wie die Unterteilung in Konkretisierung und Abstraktion, oder die Integration von Handlungen, Aufgaben oder Tätigkeiten.
3. Es wird eine Möglichkeit dargeboten, hierarchisch vorliegende Wissenskonstrukte zu zusammenhängendem Wissen zu kombinieren.
4. Das Modell bietet die Möglichkeit, automatisiert aus dem modellierten Wissen Ableitungen, Entscheidungen und andere Schlussfolgerungen zu treffen, zum Beispiel durch Induktions- und Deduktionsalgorithmen.
5. Die aufgezeigte Methode liefert eine Möglichkeit Datenerfassung oder Nutzereingaben, beispielsweise zur Erfassung sozialer Einflussfaktoren, zu modellieren und in die Schlussfolgerungsprozesse zu integrieren.

Knowledge Engineering in Verbindung mit Softwareentwicklung erfährt im Allgemeinen ein breites Anwendungsfeld. Dies wird besonders durch die zahlreich publizierten Schriften auf diesem interdisziplinären Gebiet deutlich. Aus insgesamt über 1000 Treffern in den erwähnten Publikationsdatenbanken konnten jedoch lediglich 44 Veröffentlichungen als dem Thema nahe Literatur identifiziert werden (Anhang, Tabelle 21, Seite xxxviii). Diese hohe Filterrate resultiert größtenteils aus den Domänen, auf die die Publikationen abzielen. Zwar erfolgt häufig eine Auseinandersetzung mit Ontologien und deren Anwendung, jedoch kaum hinsichtlich einer Fokussierung auf Verarbeitung von Wissen über Software Engineering. Nachfolgend finden eine Begriffsabgrenzung und die Präsentation der aus der Literaturanalyse entstandenen Ergebnisse statt.

2.2 Wissen strukturieren, organisieren und nutzbar machen

Der erfolgreiche Einsatz wissensbasierter Systeme wird maßgeblich durch die Verarbeitung des enthaltenen Wissens bestimmt. Wesentliche Punkte bei der Entwicklung dieser Systeme sind die Erfassung und Strukturierung von Wissen, die Erarbeitung einer präzisen computergestützten Wissensbasis mit integrierten Inferenzmechanismen und schließlich die Dokumentation und Visualisierung aller Assoziations- und Wissenskonstrukte. Diese Punkte werden in der Literatur zu Knowledge Engineering zusammengefasst. (HOPPE 1992, S. 29–30)

Der Abschnitt beschäftigt sich mit allen für die Anwendung des Knowledge Engineerings notwendigen Voraussetzungen. Es wird zunächst eine genauere Betrachtung wesentlicher mit Knowledge Engineering verbundenen Begriffe und ihrer Bedeutung in der Praxis gegeben.

2.2.1 Begriffsbestimmung und Einordnung

Knowledge Engineering wird in der verwendeten Literatur häufig gleichbedeutend mit Knowledge Modeling verwendet. Auch im deutschen Kontext wird der Begriff Knowledge Engineering vielfach durch „Wissensmodellierung“ übersetzt. Trotzdem scheint es im aktuellen Stand der Forschung eine Abgrenzung beider Begriffe zu geben. Was genau ist also Knowledge Engineering und wie kann es von Knowledge Modeling abgegrenzt werden?

Knowledge Engineering

Knowledge Engineering dient neben einer effektiven Strukturierung vor allem der Kommunikation und Wiederverwendung von Wissen. Durch Methoden wie Ontology Modeling können Informationen, Daten und deren Zusammenhänge in Wissensbasen einheitlich abgebildet und digital nutzbar gemacht werden. MATTA ET AL. (2002) und GUAGLIANONE ET AL. (2011) unterteilen Knowledge Engineering in Acquiring und Modeling zur gezielten Entwicklung von Experten Systemen und um Problemstellungen des Wissensmanagements zu lösen. HALL (2012) beschreibt, wie Wissen mithilfe des Knowledge Engineerings in Computersysteme integriert werden kann. CHAN (2002) betrachtet Knowledge Engineering in als Prozess, bei dem Expertwissen sichtbar gemacht und computergestützt organisiert wird, um digitale Wissensbasen zu schaffen. Dabei nimmt sich Knowlegde Engineering verschiedenen Methoden der menschlichen Informationsverabreitung an. CHAN (2002) erwähnt in seinem Artikel weiterhin: „Ontology Engineering kann als Nachfolger des Knowlege Engineering betrachtet werden“. Es verfolgt in höhrem Maße das Ziel einer langzeitig effizienten Gestaltung von Wissensbasen, auch in verteilten Systemen.

Ontology Engineering

Unter Ontology Engineering ist die Entwicklung wissensbasierter Systeme zu verstehen, deren Wissensbasis ontologiebasiert modelliert wurde. Bezugnehmend auf BREWSTER UND WILKS (2004) sowie HEPP (2007) sind Ontologiemodelle ein mit einem spezifischen Ziel erstelltes technisches Hilfsmittel, um umfangreiche deklarative Beschreibungen von Entscheidungsmodellen zu entwickeln. Diese sollen falsche Schlussfolgerungen (Inferenzen) weitestgehend verhindern und sinnvolle, automatisch erzeugte Interpretationen des integrierten Wissens ermöglichen. Die größte Herausforderung dabei bildet eine zeitlich stabile und langfristige Nutzung von Ontolgien. Zwar existieren eine Reihe von Werkzeugen zur Entwicklung von Ontologien, doch fehlt bislang im Bereich des Ontology Engineering ein Model, um zuverlässig Schlussfolgerungen skalierbar zu integrieren. Der Grund hierfür liegt im Charakter der Ontologien an sich. Sie stellen ein Werkzeug zur semiformalen Wissensrepräsentation dar und bieten bei umfangreichen und sich veränderndem Wissen kaum Möglichkeiten, eine ausreichend konsistente Formalisierung und Integrität des Modellierens zu erreichen. (HEPP 2007)

Knowledge Modeling

Nach CHAN ET AL. (2002) werden Knowledge Acquisition und Knowledge Modeling zur Erfassung und Strukturierung von Wissen sowie zur Formalisierung von Wissensbasen genutzt. Bezugnehmend auf KALANA MENDIS ET AL. (2007) bezieht sich Knowledge Modeling auf Sprachen, Werkzeuge, Techniken und Methoden zur Entwicklung abstrakter Modelle einer spezifischen Domäne oder Problemstellung. Werkzeuge und Methoden des Knowledge Modeling, die zur Strukturierung und Formalisierung von Wissen dienen, sind beispielsweise Semantische Netze, Topic Maps oder Ontologien. Von RAMIREZ UND VALDES (2011) wird beschrieben, wie Knowledge Modeling auch von kognitiven Systemen zur intelligenten Personalisierung genutzt wird. Er demonstriert Assoziationsmöglichkeiten zwischen bereits vorhandenem und zukünftig erlerntem Wissen, in dem er Basiselemente des semantischen Netzes nutzt.

Abb. 2 stellt die Begriffe Knowledge Modeling, Knowledge Engineering und Ontology Engineering und ihre Zusammenhänge grafisch gegenüber.

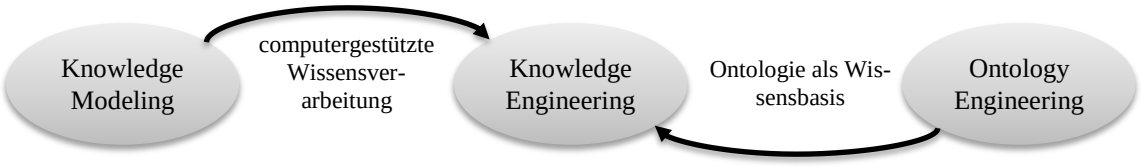


Abb. 2 Knowledge Modeling, Knowledge Engineering und Ontolgy Engineering

Die Modellierung von Wissensbasen durch Knowledge Modeling und deren computergestützte Verarbeitung durch Knowledge Engineering dienen dem gemeinsamen Zweck, Wissen zu verteilen und für unterschiedliche Anwendungsfälle anwendbar zu machen. Der Unterschied von Knowledge Modeling und Knowledge Engineering liegt demnach in der Art und Weise ihrer praktischen Verwendung. Während sich Knowledge Modeling mit Methoden und abstrakten Modellen zur Beschreibung von Wissen beschäftigt, dient Knowledge Engineering der computergestützten Verarbeitung von Wissen. Dabei verwendet Knowledge Engineering teilweise Knowledge Modeling Werkzeuge, wie beispielsweise Ontologien, was unter Umständen zu neuen Anwendungsfällen wie dem Ontology Engineering führen kann.

2.2.2 Werkzeuge des Knowledge Modeling

Ein Semantisches Netz ist ein Darstellungskonzept, bei dem ein gering formalisierter Graph mit Knoten und Kanten genutzt wird, um Wissen und Zusammenhänge zu strukturieren. Topic Maps, die auch unter dem Begriff Knowledge Maps bekannt sind, bestehen formal aus Topics, Assoziationen und Ereignissen, den sogenannten Subjekten. Sie orientieren sich an der menschlichen Verarbeitung von Wissen. Die Beschreibung dieses Wissen erfolgt durch XML, weshalb Topic Maps einfacher lesbar für computergestützte Systeme sind. In erster Linie dienen Topic Maps zur besseren Navigation und Suche im Internet und zum Austausch von Metadaten zwischen verschiedenen Endpunkten (PEPPER 2000).

Ontologien als Methode des Knowledge Modeling bieten nach CHAN (2002) die Möglichkeit, durch Terminologien und deren Abhängigkeiten ein beliebiges Fachgebiet zu beschreiben. Dies kann genutzt werden, um Wissensbasen zu schaffen, die Zusammenhänge realer oder imaginärer Umgebungen enthalten. Daher handelt es sich bei Ontologien um eine weitestgehend formale Repräsentation von Wissen, die durch Mechanismen der Vererbung umgesetzt werden. Sie bieten die Möglichkeit einer einheitlichen Kommunikation, beispielsweise durch die Web Ontology Language. Eine erfolgreiche Anwendung im Bereich des Product Redesins wurde von LIU ET AL. (2010) vorgestellt. Hier dient ein ontologiebasierter Knowledge Modeling Ansatz zur Wiederverwendung von Wissen im Bereich Maschinenbau. Die Methode namens Ontology Modeling zeigt, wie Herausforderungen der Wissensintegration bei existierenden Wissensbasen bewältigt werden können. Ziel dieses Ontology Models ist ein effizienterer und effektiverer Einsatz von digitalem Wissen, speziell in der Umgebung der Maschinenbauindustrie.

In Anlehnung an BREWSTER UND WILKS (2004) zeigt Abb. 3 die beschriebenen Werkzeuge in Abhängigkeit ihres jeweiligen Formalisierungsgrads. Dieser ist entscheidend, um Algorithmen zu integrieren, mit deren Hilfe Schlussfolgerungen aus dem hinterlegten Wissen gezogen werden können. Mind Maps weisen aufgrund ihrer frei vom Anwender wählbaren Darstellungsmöglichkeiten von Anwendungsfall zu Anwendungsfall eine völlig unterschiedliche Struktur auf. Daher symbolisieren sie in dieser Grafik den geringsten Grad der Formalisierung.

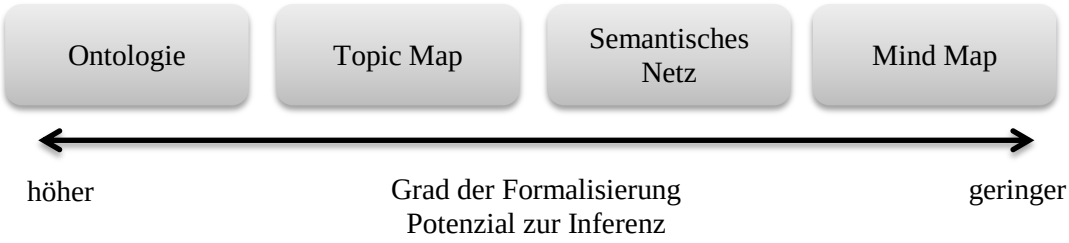


Abb. 3 Werkzeuge des Knowledge Modeling

Eine ausführlichere Beschreibung von Ontologien zur Modelierung von Wissen erfolgt in Abschnitt 2.3 ab Seite 16.

2.2.3 Wissensbasierte Systeme im Fokus des Knowledge Engineering

Unter einem wissensbasierten System wird in den meisten Fällen eine Software beschrieben, die anwendungsspezifische Entscheidungen unterstützen oder sogar selbstständig treffen kann. Basierend auf Inferenzmechanismen, einer Benutzerschnittstelle und einer Wissensbasis ist solch ein System dazu in der Lage, Lösungen für spezifische Problemstellungen hervorzubringen. Möglichkeiten zur Anwendung wissensbasierter Systeme sind vielfältig. Sie reichen von Data Mining über medizinische Überwachungs- und Diagnosesysteme bis hin zur planungsunterstützenden Anwendung im Unternehmensbereich. Als wesentlichste Vertreter wissensbasierter Systeme gelten Expertensysteme, die im Folgenden kurz vorgestellt werden.

Expertensysteme

Nach RUSSELL UND NORVIG (2010) kann ein Expertensystem beschrieben werden, als „System, das einige Komponenten menschlicher Intelligenz nachahmen kann“. Expertensysteme werden mit dem Ziel entwickelt, Wissen eines Menschen zu akquirieren und in maschinenlesbaren Code zu transformieren. Nach CHAN (2002) ist die Entwicklung eines Expertensystems „ein wissensintensiver Prozess“, bei dem Knowledge Engineering und Methoden des Knowledge Modeling zur Erarbeitung der Wissensbasis eingesetzt werden.

Um ein System auf Basis künstlicher Intelligenz zu schaffen, muss zunächst festgestellt werden, wie intelligentes, menschliches Verhalten von einem Computersystem nachgebildet werden kann (HWANG ET AL. 2011; KULINICH 2012). Dazu dienen Inferenzregeln, wie Ursache-Wirkung, Zustand-Reaktion oder Wenn-Dann. Auf diese Weise können Experten-

systeme zur Problembehebung, zur Entscheidungsfindung, zum Entwerfen, Planen oder Überwachung genutzt werden (HWANG ET AL. 2011).

Menschen haben Schwierigkeiten schnellstmöglich Probleme unter Einbezug aller Einflussfaktoren zu erfassen und zu lösen, da die menschliche Wahrnehmung in diesen Fällen ihre Grenzen erreicht. Expertensysteme haben diesbezüglich einen entscheidenden Vorteil. Durch die weitestgehend freie Skalierbarkeit von Hard- und Softwarekomponenten, kann ein Expertensystem jederzeit multidimensional Wissen und Erfahrungen zur Lösung eines Problems einbeziehen. (NGUYEN UND KIRA 2000)

GUAN UND LEVITAN (2012) sehen das Potenzial zukünftiger Expertensysteme vor allem bei der Akquise und Integration von Wissen unterschiedlicher Fachbereiche. Insbesondere bei der Zusammenführung unterschiedlicher Wissensbasen besteht derzeit hohes Entwicklungspotenzial.

Lernende Expertensysteme

Kombiniert mit Wissensbasen dienen beispielsweise künstliche neuronale Strukturen oder die Repräsentation von regelbasiertem Wissen zur Entwicklung lernfähiger Expertensysteme (TIAN ET AL. 1995; FENG UND LINWEN 2008). Mithilfe dieser Komponenten wird es möglich, Expertensysteme mit einer trainierbaren Wissensbasis auszustatten. Eine Kombination aus Nutzereingaben und akquirierten Wissen führt zu neuen Wissenskonstrukten, die von Inferenzmechanismen zu sinnvollen Schlussfolgerungen verarbeitet werden können. Auf eine genaue Beschreibung von Funktionsweise und Aufbau künstlicher neuronaler Netze wird im Rahmen dieser Arbeit verzichtet.

Abb. 4 verallgemeinert die Idee, Expertensysteme mit Trainingskomponenten zu kombinieren. Die Abbildung demonstriert auf einfache Weise die inneren und äußeren Komponenten eines wissensbasierten Systems unter Einbezug von neuronalen Strukturen und symbolischer KI[1]. Darüber hinaus zeigt die Grafik den Zusammenhang von Knowledge Engineering, Knowledge Modeling und wissensbasierten Systemen. Wie in der Darstel-

[1] Ansatz der Künstlichen Intelligenz zur Repräsentation und Verarbeitung von Wissen und Informationen

lung zu sehen, bestehen wissensbasierte Systeme aus einem Inferenzsystem und einer Wissensbasis.

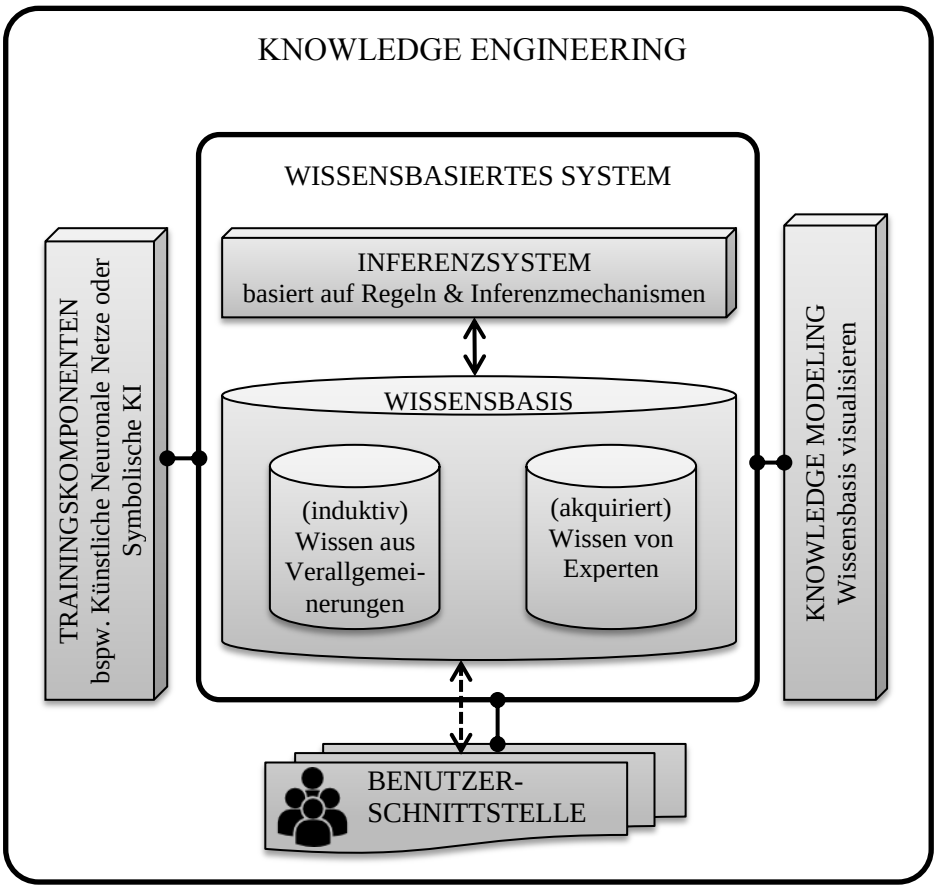


Abb. 4 Wissensbasierte Systeme im Fokus des Knowledge Engineering

Basierend auf Regeln und Inferenzmechanismen kann das Inferenzsystem mit der Wissensbasis kommunizieren. Zur Erstellung von Wissensbasen dienen die Werkzeuge des Knowledge Modeling. Die Wissensbasis als solche setzt sich zusammen aus Wissen von Experten, das manuell in das System eingepflegt wird und Wissen, welches aus der Verallgemeinerung von Beispielen induktiv erworben wird. Letzteres ist vom System eigenständig generiertes Wissen, das zum Beispiel durch symbolische KI als Trainingskomponente beeinflusst wird. Zur Repräsentation des Wissens dient eine Benutzerschnittstelle. Diese kann einerseits Eingaben zum langfristigen Speichern von Expertenwissen entgegen nehmen, anderseits dient sie als Ausgabe entsprechender Ergebnisse nach Beendigung einer gezielten Problem- oder Fragestellung.

Wie in den vorangegangenen Absätzen beschrieben, bilden Methoden zur Schlussfolgerung (Inferenz) und eine geeignete Wissensbasis die Grundlage wissensbasierter Systeme.

Bei der Entwicklung dieser Systeme bildet die Diskrepanz zwischen menschenlesbarer Darstellung des enthaltenen Wissens und der Erstellung von Wissen aus maschinellem Lernen eine der größten Herausforderung. Ein Modell, welches diese Lücke durch standardisierte Zusammenführung von Ontologien und Trainingsansätzen schließt, wäre ein geeigneter Ansatz zur Zielerfüllung dieser Arbeit.

2.3 Ontologie als formalisiertes Werkzeug des Knowledge Modeling

Wie bereits im vorangegangenen Abschnitt beschrieben, bildet die Integration von Wissen die Basis zur Entwicklung eines intelligenten Systems. Die ganzheitliche Systementwicklung kann als Knowledge Engineering beschrieben werden. Knowledge Engineering orientiert sich dabei am Prozess des Software Engineering (Anforderungsmanagement, Softwarearchitektur, Implementierung, etc.), richtet den Fokus dabei jedoch auf die Entwicklung von wissensbasierten Systemen. Neben der Betrachtung aller erforderlichen Ingenieurleistungen, ist es vor allem notwendig, die beiden Kernkomponenten eines wissensbasierten Systems zu erarbeiten. Diese sind das Inferenzsystem zur Abbildung von Regel- und Inferenzmechanismen und die Wissensbasis, welche Experten- und Erfahrungswissen in strukturierter Form enthält. Die Modellierung einer Wissensbasis, also das Strukturieren und Visualisieren von Expertenwissen, wird als Knowledge Modeling bezeichnet. Für das Knowledge Modeling stehen derzeit verschiedene Werkzeuge zur Verfügung (vgl. Abschnitt 2.2.2). Abschnitt 2.3 beschäftigt sich ausführlich mit dem derzeit bekanntestem Werkzeug des Knowledge Modeling: den Ontologien.

2.3.1 Nutzen, Anwendungsbereiche und Aufbau von Ontologien

Ihren Ursprung fanden Ontologien bereits zu Beginn der Frühmoderne in der Lehre der Philosophie. Hier dienen sie bis heute zur Beschreibung von Wirklichkeit und Sein, was sie zu einem Synonym der Metaphysik werden lässt (ZIMMERMANN 1998). Durch sprachliche Darstellungen von Zusammenhängen in Natur und Wirklichkeit verkörpern Ontologien auch aus philosophischer Sicht Wissen. Diese Eigenschaft, Wissen zu verkörpern, haben sie mit der in der Informatik üblichen Begriffsverwendung gemeinsam. Doch bezeichnet der Begriff Ontologie in der Informatik im Gegensatz zur Philosophie keine Lehre oder Disziplin, vielmehr dient er der Benennung eines Wissensmodells. Mit anderen Worten: Aus Sicht der Informatik existiert nicht die eine Ontologie, sondern je nach enthaltenem Wissen eine Vielzahl von Ontologien.

Die Informatik verwendet Ontologien zur formalen Repräsentation von Wissen. Es werden sowohl grafische als auch formal-sprachliche Hilfsmittel genutzt, um Begriffen Objekte zuzuordnen. Neben der Integration dieser Begriffe werden in Ontologien auch deren Zusammenhänge verdeutlicht. In den meisten Fällen beziehen sich Ontologien auf einen konkreten Gegenstandsbereich, also ein spezielles Fachgebiet. (GRUBER 1995)

Wofür und wie Ontologien verwendet werden, wird in den folgenden Absätzen ausführlicher beschrieben. Sie gehen auf Anwendungsbereiche und Bestandteile von Ontologien ein, sowie auf verschiedene Typen und Sprachen dieses Modellierungswerkzeuges. Insbesondere soll der Grad der Formalisierung genauer betrachtet werden, da dieser als Basis zur maschinellen Verarbeitung dient. Begonnen wird zunächst mit der Fragestellung nach dem eigentlichen Nutzen und den Zielen von Ontologien.

Nutzen und Ziele von Ontologien

Das wohl bedeutendste Ziel von Ontologien kann der Repräsentation von Wissen zugeschrieben werden. Dabei dienen sie nicht allein der Visualisierung mentaler Modelle. Vielmehr wird Wissen mithilfe von Ontologien strukturiert. Abstraktionen können geordnet, verschiedene Zusammenhänge dargestellt und zu Konkretisierungen aufgelöst werden. Diese Eigenschaft von Ontologien ermöglicht es Ausschnitte der Realität abzubilden. JANSEN ET AL. (2008) sprechen in diesem Zusammenhang von Realitätsrepräsentation.

Durch Ontologien liegt Wissen explizit, vielfach sogar digitalisiert vor, was zu einem weiteren Nutzen dieser Modellierungsmethode führt: der einfachen Wiederverwendung von Wissen. Durch „formale und maschinenlesbare Definition einer Domäne“ können modellierte Informationen jederzeit nutzbar gemacht werden (DEGLE 2007). Ein weiterer Vorteil der formalen Definition liegt in der Erkennung von Abweichungen zwischen modelliertem Wissen und Ist-Zuständen der Realität. Daher können Ontologien als Datenbasis für Softwaresysteme dienen, die Ist-Daten mit Sollwerten vergleichen, um Fehler und Störungen zu erkennen und zu vermeiden. Weiteres wesentliches Ziel der formalen Definition in Ontologien ist Interoperabilität. Ontologien ermöglichen den Austausch und die Kommunikation von Wissen zwischen Mensch und Maschine und dienen somit als Basis zur automatisierten Schlussfolgerung in intelligenten Systemen.

Nicht zuletzt erhalten Ontologien in den letzten Jahren besondere Aufmerksamkeit durch das Semantische Web. Ein aus ihnen hervorgegangenes Werkzeug mit dem Ziel, Inter-

netressourcen maschinenlesbar zu gestalten. Dies kann beispielsweise zur Prozessoptimierung im E-Commerce beitragen indem intelligente Agenten automatisiert Kaufentscheidungen treffen und direkt die Auftragsabwicklung durchführen.

Anwendungsbereiche

Wie bereits erwähnt werden Ontologien im Knowledge Engineering zum Knowledge Modeling angewandt. Hierbei sollen sie zur Wissensrepräsentation dienen, da sie alle notwendigen Informationen in einem Modell integrieren. Dazu zählen neben der Integration einfacher Daten, vor allem Unternehmensinformationen, Umweltausschnitte, Prozessabbilder oder softwarerelevante Artefakte.

Da Wissen in Ontologien langfristig strukturiert vorliegt, erhöht dies die Möglichkeit Informationen auch nach langer Zeit wieder aufzufinden. Aus diesem Grund wird Ontologien auch im betrieblichen Wissensmanagement hohe Aufmerksamkeit zuteil, beispielsweise im Bereich der semantischen Suche. Sie dienen Anwendern und Anwendungen gleichermaßen, da sie als gemeinsame Wissensbasis einer bestimmten Thematik zur Problemlösung herangezogen werden können.

Darüber hinaus werden Ontologien eingesetzt, um Konzepte und Datenbankentwürfe zu modellieren. Sie können zwischen Beschreibung und Instanzen unterscheiden, was sie zu einem interessanten Werkezeug für die Architekturbeschreibung von Softwaresystemen werden lässt. So können Softwarekomponenten, Klassen und Entitäten zunächst abstrakt in einer beschreibenden Ebene von Ontologien integriert werden, die anschließend als konkrete Objekte instanziiert werden können.

Populäre Anwendungen von Ontologien ergeben sich aus der Möglichkeit, Sprachverarbeitung zu unterstützen. In zahlreichen Projekten erfolgt dies durch die vollständige Eingliederung von WordNet, einer linguistischen Ontologie um natürlich-sprachliche Texte maschinell verarbeitbar zu machen. Dabei handelt es sich um eine Ontologie, die semantische und lexikalische Zusammenhänge zwischen verschiedenen Wörtern enthält (MILLER ET AL. 1990). Im Laufe der Zeit wurde WordNet um Synonyme, Akronyme und Hyperonyme erweitert.

Weitere Anwendungsbereiche von Ontologien ergeben sich in der Biologie, der Anthropologie, dem E-Commerce, der Bioinformatik, der Geosemantik, verschiedenen unternehme-

rischen Anwendungen und in der Medizin (JANSEN ET AL. 2008). Vor allem in der Medizin wird die Thematik der Ontologien intensiv genutzt. Das sogenannte Unified Medical Language System (UMLS) ist eine Ontologie zur computergestützten Kommunikation und Analyse menschlicher Krankheiten oder Symptome. Die Erarbeitung dieser Ontologie begann bereits 1986 und wurde später von LINDBERG ET AL. (1993) vorgestellt. Seither erfährt diese Ontologie zunehmend Popularität und Nutzung.

Komponenten einer Ontologie

Ontologien bestehen im Wesentlichen aus Begriffen, Instanzen und Relationen (Abb. 5).

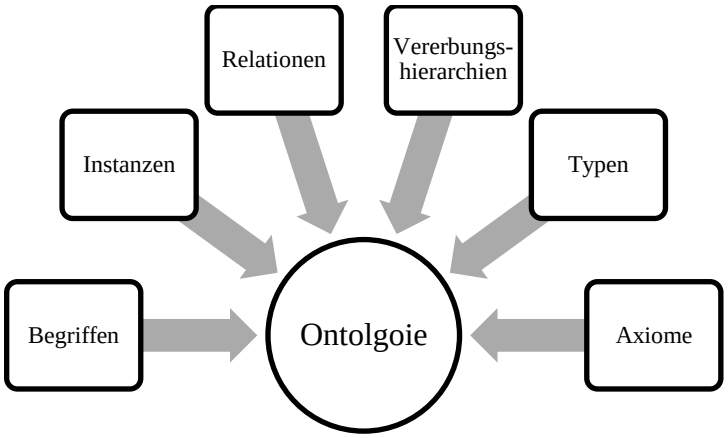


Abb. 5 Komponenten einer Ontologie

Begriffe, auch Konzepte (engl. concepts) oder Schlüsselkonzepte bezeichnet, sind aus objektorientierter Sicht Klassen von Objekten gemeinsamer Eigenschaften (HAARMANN 2014). Ein Beispielkonzept ist *Frucht*. Jede Frucht besitzt eine Farbe. Gelb ist eine Farbe, weshalb auch Farbe eine Klasse ist. Banane ist eine Frucht, sie hat die Farbe Gelb. Banane ist ein konkretes Objekt, eine Instanz der Klasse Frucht.

Neben Begriffen existieren demnach auch Instanzen in Ontologien. Diese repräsentieren konkrete Einzelobjekte einer Klasse oder eines Types. Typen (types) sind in Ontologien Verfeinerungsstufen von Klassen, die ihre Eigenschaften an konkrete Instanzen vererben können. Neben Begriffen, Typen und Instanzen werden in Ontologien Beziehungen verwendet, um Zusammenhänge zwischen den Komponenten darzustellen. Dies kommt den Relationen aus dem Entity-Relationship-Model sehr nahe. Alle Objekte weisen Eigenschaften auf, die auch als Attribute bezeichnet werden, beispielsweise ist Farbe eine Eigenschaft von Früchten. Ein weiteres Merkmal von Ontologien ist die Möglichkeit Klassen

und Objekten Eigenschaften einer anderen Komponente zu vererben. Komponenten können dabei auch von verschiedenen Objekten abgeleitet sein (Mehrfachvererbung). Die Vererbung wird auch als Ableitung oder Delegation bezeichnet. Es wird nach zwei verschiedenen Hierarchierichtungen unterschieden. Bei der Bottom-Up-Hierarchie treten transitive Vererbungsstrukturen auf. Beispielsweise erbt MS Word alle Eigenschaften einer Software, wenn folgende Vererbungsstruktur vorliegt: MS Word ist Textverarbeitung. Textverarbeitung ist Software. Die zweite Hierarchierichtung ist die Top-Down-Hierarchie, bei der das oberste Objekt nicht die Eigenschaften der unteren Hierarchieebenen erbt. (HAARMANN 2014)

Ontologien werden in leichtgewichtige (leight-weighted) und schwergewichtige (heavy weighted) Ontologien unterschieden. Im Unterschied zur leichtgewichtigen Ontologie, die ausschließlich Begriffe, Vererbungshierarchien und Relationen enthält, werden schwergewichtige Ontologien um Axiome ergänzt. Diese Axiome können als Ausdrücke oder Aussagen verstanden werden, die ergänzendes Wissen zweier in Beziehung stehender Objekte beinhalten. Axiome können zur Plausibilitätsprüfung herangezogen werden, da diese Ausdrücke immer einer wahren Aussage entsprechen, beispielsweise „Die Erde ist nicht Mittelpunkt unseres Sonnensystems“. (COLLARD 2007, S. 38)

2.3.2 Typisierung von Ontologien

Ontologien können auf Basis zweier Betrachtungsgrade klassifiziert werden: dem Genauigkeitsgrad und dem Abhängigkeitsgrad (GUARINO 1997, S. 139–170). Dabei werden sie im Allgemeinen in vier Ontologietypen unterteilt.

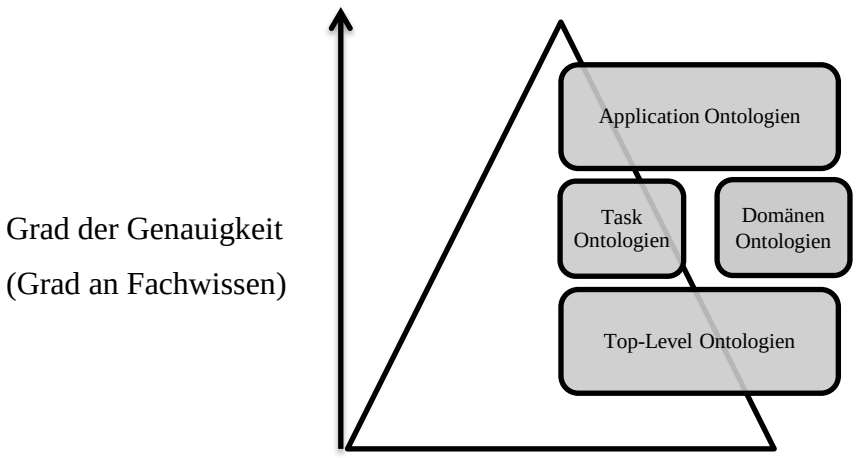


Abb. 6 Ontologietypen nach ihrem Anteil an Fachwissen (nach DEGLE 2007, S. 6)

Abb. 6 zeigt diese Ontologietypen in Abhängigkeit des Genauigkeitsgrades und der Anwendungsbreite. Während Top-Level Ontologien eine breite Anwendung in zahlreichen fachspezifischen ontologiebasierten Systemen finden, können spezielle Application Ontologien nur für eine begrenzte Anzahl an Anwendung genutzt werden. Der Grad der Genauigkeit symbolisiert dabei das in der Ontologie enthaltene integrierte Fachwissen.

Top-Level-Ontologien

Ontologien, mit dem geringsten Spezifikationsgrad, werden als Top-Level-Ontologien bezeichnet (auch Upper- oder Foundation-Ontologien). Sie enthalten Wissen über allgemeine Begriffe und Konzepte und werden als Basis zur Charakterisierung von neutralem, unspezifischem und allgemeingültigem Wissen verwendet. Einige Ontologiespezialisten, wie Barry Smith, Nicola Guarino oder Heinrich Herre verfolgen die Vision, dass weltweit nur eine Instanz dieses Ontologietypes erforderlich wäre. Top-Level-Ontologien enthalten Beschreibungen allgemeiner Terminologien und deren Zusammenhänge, wie Informationen über Zeit, Raum, Objekte, Fakten oder Situationen. Sie können daher als Grundlagenwissen für zahlreiche Domänen genutzt werden. (DEGLE 2007, S. 6; ARP ET AL. 2015)

Vorteile von Upper-Level-Ontologien liegen insbesondere in der Wiederverwendbarkeit ontologiebasierter Wissensmodellierung. Dieser Ontologietyp dient einer generischen Verwendung von Allgemeinwissen und ist daher zur freien Verwendung in spezifischen Domänen geeignet. Nach HOEHNDORF (2010) sind die meist genutzten Upper-Level-Ontologien: BFO, DOLCE und SUMO. Sie werden im Folgenden kurz vorgestellt.

Die BFO (Basic Formal Ontology), die vom Onto-Med der Universität Leipzig entwickelt wurde, ist eine sehr formale Top-Level-Ontologie mit dem Ziel der Integration von Informationen zwischen verschiedenen Fachbereichen. Hierbei werden im Wesentlichen zwei Konzepte unterschieden: Continuant und Occurent. Während Continuant ausschließlich zeitlich unabhängige Begriffe beschreibt, wie Mensch oder Festplatte, beschreiben Occurents Begriffe mit zeitlichen Abhängigkeiten, wie bestimmte Prozesse oder Events. Durch ihren modularen Aufbau können verschiedene Wissensmodellierer an derselben Ontologie arbeiten. Dies vereinfacht die Wiederverwendung und die Prüfung des enthaltenen Wissens. BFO enthält keine Unterstützung für mathematische Daten, wie Zahlen oder Definitionen und keine Möglichkeit Boolsche Ausdrücke zu integrieren. Weiterhin wird durch BFOs nicht das Ziel verfolgt, spezifisches Domänenwissen zu modellieren, sondern

domänenneutrale Informationen zu strukturieren und digital verfügbar zu machen. (ARP ET AL. 2015)

Hauptanwendungsbereiche der BFO sind in der Bioinformatik und in der Medizinischen Technik zu finden. Hier wird die BFO in zahlreichen Ontologien als Top-Level-Ontologie eingesetzt. Beispiele dieser Projekte sind OncoCl (Krebszellen Ontologie), BioTop (biomedizinische Top-Level-Ontologie), BLO (Blut), EO (Ontologie mit Evolutionsdaten) oder die HDOT (Ontologie zur Personalisierung in der Medizin). Durch die Integration von „Information Artifacts" wird neben der Medizin auch in weiteren Anwendungsbereichen BFO eingesetzt. Somit sind Ontologien wie SWO (Software Ontology), OntoDM (Ontology of Data Mining), Computational Neurosience Ontology, Twitter Ontology oder die Population and Community Ontology entstanden.

Eine weitere vielseitig eingesetzte Upper-Ontologie ist DOLCE (Descriptive Ontology for Lingustic and Cognitive Engineering). Sie enthält neben allgemeinem Wissen als Top-Level-Ontologie auch Domänenwissen und legt ihren Fokus auf Linguistik mit dem Ziel kognitiven Systemen eine hinreichende Wissensbasis zugrunde zu legen. Entwickelt wurde diese Ontologie im Rahmen des „WonderWeb foundation ontologies library"-Projektes des Laboratory for Applied Ontology. Hauptmerkmal dieser Ontologie bilden Beschreibungen, deren Gebrauch in der natürlichen Sprache in Abhängigkeit von kulturellen und sozialen Faktoren steht. (ROBINSON 2011)

DOLCE unterscheidet ähnlich der BFO Endurants, also zeitlich unabhängige Objekte, und Perdurants mit zeitlichem Bezug. Zur besseren Unterscheidung beider Klassifikationen können Begriffe auf ihre zeitliche Vergänglichkeit überprüft werden. Endurants sind Begriffe, die zu jeder Zeit existieren, sich während dieser Zeit jedoch ändern können. Als Beispiel für Endurants können physisch existierende Objekte genannt werden, wie Haus oder Baum. Perdurants hingegen sind durch ein Start- und ein Endzeitpunkt charakterisiert. Sie existieren also nur partiell für den Zeitraum, in dem sie sich befinden. Diese Art von Klassen können Events oder Prozesse darstellen, was sie zu Artverwandten der Occurants einer BFO macht. Sowohl Endurants als auch Perdurants können Bestandteile aufweisen, wie die Finger einer Hand oder das Fell des Löwen. Der Hauptunterschied zur BFO liegt in der Integrationsmöglichkeit physischer Objekte, bei denen es weitere Unterscheidungsebenen gibt, wie Aggregate, Merkmale oder Lebendigkeit. Ähnlich zu den Vererbungsstrukturen der BFO existieren auch in DOLCE Topologien, die es ermöglichen Merkmale oder

Eigenschaften einer Elternklasse an Kind- und Kindeskindklassen weiterzugeben. Durch die Möglichkeit der formalen Beschreibung von Abhängigkeiten zu Eigenschaften oder anderen Objekten ist es möglich in DOLCE das enthaltende Wissen zu prüfen. Die dabei modellierten Abhängigkeitsstrukturen, die zur Überprüfung real existierender Objekte herangezogen werden, befinden sich jedoch auf einem sehr hohen Abstraktionsgrad. Es existieren allgemein gültige Validierungsregeln, wie „alle Instanzen eines bestimmten Objektes haben ein sie eindeutig identifizierbares Merkmal“. (MASOLO ET AL. 2003)

Anwendungsprojekte von DOLCE sind ODASP (Ontology Driven Analysis of Nominal Systematic Polysemy in WordNet), ILIKS (Interdisciplinary Laboratory on Interacting Knowledge Systems), MOSTRO (Modelling Security and Trust Relationships within Organizations), OntoWeb (Ontology-based information exchange for knowledge management and electronic commerce) oder FOS (Fishery Ontology Service).

Auch aus der Nutzung der Suggested Upper Merged Ontology (SUMO) ergeben sich zahlreiche Anwendungsfelder, die von NILES UND PEASE (2001) erstmals vorgestellt wurde. SUMO enthält mathematische Komponenten, was sie zu einer deutlich umfangreicheren Ontologie macht, als beispielsweise BFO. Ziele sind eine natürliche Sprachverarbeitung und Information Retrieval. Sie besteht aus Mengen und Graphenkomponenten und beinhaltet Zeitkonzepte, Vorgänge, Zahlen und Maße. Die SUMO wird beispielsweise in der Elektrotechnik verwendet, um verteilte und heterogene Sensordaten zu suchen. Dabei wird SUMO genutzt, um Basisbegriffe und ihre Zusammenhänge zu modellieren (EID ET AL. 2007).

Domain- und Task-Ontologien

Die Domain-Ontologie weist im Vergleich zur Top-Level-Ontologie einen höheren Grad an spezifischem Wissen auf und verfolgt daher gezielt die Wissensintegration eines speziellen Anwendungsbereichs. Sie beschreibt das Vokabular einer spezifischen Domäne, wie Medizin, Nano- oder Softwaretechnik. Task-Ontologien enthalten domänentypische Beschreibungen bestimmter Abläufe oder Tätigkeiten. Dies wird realisiert, indem die Begriffe und Zusammenhänge einer Top-Level-Ontologie spezifiziert werden, um das Wissen eines konkreten Anwendungsbereiches zu modellieren. (DEGLE 2007)

Da Domänen- und Task-Ontologien spezifisches Fachwissen enthalten, was sie zu speziellen Wissensspeichern werden lässt, beinhalten sie idealerweise alle relevanten Terminolo-

gien eines Fachgebietes. Aufgrund der Fachgebietsvielfalt unseres Gesellschaftssystems existieren entsprechend vielfältige Anwendungsmöglichkeiten dieser Ontologietypen. Beispielsweise haben MIZOGUCHI ET AL. (2000) in ihrer Veröffentlichung eine Domänen-Ontologie für Pflanzen vorgestellt, in der sie Wissen über Pflanzen durch Labels, Axiome, Definitionen und Taxonomien strukturiert haben. Das am weitesten verbreitetste Anwendungsgebiet von Domänen-Ontologien sind Medizin und Bioinformatik. Von KHELIF UND DIENG-KUNTZ (2004) wird eine Domänen-Ontologie vorgestellt, die das Ziel einer ontologiebasierten Notation für Biochip-Experten verfolgt. Eine Domänen-Ontologie für medizinische Terminologien, die zeitliche Entwicklungen von Krankheiten sowie Diagnoseverfahren zur Behandlung von Patienten beinhaltet, bieten PALMA ET AL. (2006) Sie zeigen in ihrer Publikation ausführlich die Integration zeitlich veränderbarer Wissenskonstrukte und wie sich diese durch Historien in die Ontologien einbinden lassen. Auf dem Gebiet „Kollaborative Innovation“ wurde in einem Beitrag zur Integration von Wissen in Innovationsnetzwerken von HIRSCH ausführlich ein ontologiebasiertes Entscheidungsunterstützungssystem vorgestellt.

Schnittmengen und Überschneidungen von Terminologien verschiedener Fachgebiete stellen die größte Herausforderung der Domänenmodellierung dar. Der Begriff „Ebene“ einer Top-Level-Ontologie könnte in einer Domänen-Ontologie für Topografie eine geografische Bezeichnung einer Landschaft beschreiben, während er in einer Grafikbearbeitungs-Domäne verschiedene Schichten eines Bildes darstellen würde. Da diese Art von Ontologien spezifisches Fachwissen enthalten, sind sie häufig nicht miteinander kompatibel, was die Zusammenführung zweier (oder mehr) Domänen-Ontologien erschwert. Domänen-Ontologien, die auf derselben Ontologiestruktur basieren, lassen sich leichter automatisch integrieren. Dazu existieren bereits Versuche und Projekte, wie das an der Universität von Edinburgh entwickelte Verfahren ORS (MCNEILL UND BUNDY 2007).

Application-Ontologie

Den speziellsten Ontologietyp bilden die Application-Ontologien. Sie enthalten Beschreibungen, denen in Abhängigkeit von einer Domänen- und einer Task-Ontologie meist spezifische Komponenten einer Softwarearchitektur zugeordnet werden (DEGLE 2007). Diese von GUARINO (1997) erstellte Kategorie von Ontologien ist vergleichbar mit konzeptionellen Modellen zur Klassen- oder Datendefinition einer Softwareanwendung. Dabei verfolgen Application Ontologien das Ziel, anwendungsbezogene Use Cases in die Wissensbasis

zu integrieren. Beispiele für Application Ontologien sind die EFO (Experimental Factor Ontology) auf dem Gebiet der Genforschung oder das NIFSTD, eine auf Basis des Neuro-Informatics Framework (NIF) aufgebaute Ontologie der Neuroforschung. NIFSTD enthält beschreibungen zu Anatomie, Zellen, Zellkulturen, Molekülen und Funktionen und Dysfunktionen. HARRISON UND CHAN (2005) haben ein Decision Support System untersucht, das Managern und Ingenieuren bei der Beseitigung schädlicher Materialien in der Petroleumindustrie unterstützen soll. Die dafür erforderliche Wissensbasis wurde mithilfe einer Application Ontologie umgesetzt.

2.3.3 Ontologiesprachen

Bei der Erstellung von Ontologien werden Modellierungskonzepte benötigt, die auf formaler Beischreibungslogik aufbauen. Nur so können eine maschinelle Lesbarkeit der Ontologie ermöglicht und alle Vorteile einer IT-gestützten Datenverarbeitung genutzt werden. Um das in der Ontologie enthaltene Wissen zwischen verschiedenen IT-Systemen austauschen und verarbeiten zu können, wird es mithilfe sogenannter Ontologiesprachen in eine einheitliche syntaktische Form überführt. Der Vorteil solcher Austauschformate liegt insbesondere in der Möglichkeit, verteiltes Wissen in unterschiedlichen Systemen verfügbar zu machen, um beispielsweise Techniken zur Schlussfolgerung oder Wissensergänzung von der Wissensrepräsentation zu entkoppeln.

GÓMEZ-PÉREZ ET AL. (2004) unterteilen Ontologiesprachen in zwei Gruppen: Traditional Ontology Languages und Ontology Markup Languages. Letztere werden explizit für den Einsatz von Ontologien durch Internettechnologien herangezogen. Im Unterschied zu den traditionellen framebasierten, netzwerkartigen und objektorientierten Ontologiesprachen basieren die Ontology Markup Languages auf XML und enthalten sowohl grafische als auch textuelle Syntax und Semantik. Wesentliche Vertreter der Ontology Markup Languages sind beispielsweise DAML+OIL, SHOE und OWL. Zu den traditionellen Ontologiesprachen zählen LOOM, OCML oder FLogic. (KARAKOL 2016; GÓMEZ-PÉREZ ET AL. 2004)

Die Basis von Ontology Markup Languages bilden sogenannte Beschreibungslogiken (engl. Description Logic, kurz DL), die häufig als Formalismen zur expliziten und impliziten Wissensrepräsentation beschrieben werden. Ziel dieser DLs ist die Abwägung aus schlanker Formalisierung und ausdrucksstarker Syntax, um menschliche Anwendbarkeit

mit der Erfassung komplexer Wissensstrukturen zu kombinieren. Auf Basis sogenannter Konstruktoren können mit DLs weniger komplexe oder atomare Konzepte zu komplexen Beschreibungen überführt werden. Die Anzahl an Konstruktoren wird dabei als Ausdrucksmächtigkeit bezeichnet. Sie beeinflusst maßgeblich die Komplexität einer Beschreibungslogik und damit den Grad der Anwendbarkeit automatischer Schlussfolgerungen (Automated Reasoning). (KARAKOL 2016)

Mit Hilfe der OWL 2 DL *SHROIQ(D)* lassen sich beispielsweise Beschreibungslogiken in die Ontologiesprache OWL integrieren. Die Buchstaben *SHROIQ(D)* stehen dabei für die Konstruktoren, welche von der Beschreibungslogik unterstützt werden. Diese umfassen die Beschreibungssprache ALC, welche unter anderen Konjunktionen, Restriktionen, Quantifizierer, Top und Bottom-Prinzipien, Negation oder Disjunktion enthält und unter Einbezug von Transitivität den Buchstaben S darstellen. Die weiteren Buchstaben stehen für Hierarchien (H), Funktionen/Rollen (R), Konzepte (O), Umkehrfunktionen (I), nummerische Restriktionen mit Merkmalsbezug (Q) und Datentypen (D).

DLs bestehen im Wesentlichen aus einer logikbasierten Semantik, die in die aus der künstlichen Intelligenz bekannten terminologischen und deklarativen Komponenten (Tbox und Abox) eingeteilt werden kann. Während in der Tbox Klassen und Beschreibungen, also Abstraktionen enthalten sind, beinhaltet die Abox Instanzen und Ausprägungen dieser Elemente. Drei der bekanntesten Vertreter von ontologiebasierten Beschreibungssprachen sind DAML+OIL, RDF und OWL. Sie werden im Folgenden kurz vorgestellt. (AHMAD 2013, S. 304–305)

RDF

Als Grundlage verschiedenerer Description Languages dient RDF (Resource Description Framework). Ursprünglich diente RDF als Kerntechnologie, um Metadaten in Webanwendungen einbinden und verteilen zu können. Dazu wird damals wie heute XML als Formalisierungssyntax verwendet. Als Ressourcen kommen alle Artefakte infrage, die eine eindeutige Identifizierbarkeit aufweisen, beispielsweise durch URLs, ISBNs oder DOIs. Die Beschreibungen dieser Ressourcen erfolgt unter anderen auf Basis ihrer Eigenschaften, ihres Nutzens oder ihrer Zielgruppe. RDF liefert dabei ein Model (Framework), welches die Vielfältigkeit der Beschreibungsmöglichkeiten vereinheitlicht und somit maschinenlesbar aufbereitet. RDF kann zur Beschreibung von Artikeln in Online-Shops, zur Planung von

Web-Events, zur Beschreibung von Webseiten (Inhalt, Autor, Datum, etc.), zur Beschreibung von Bildern, zur Platzierung von Informationen in Suchmaschinen oder für elektronische Bibliotheken genutzt werden. (w3schools 2016)

Wie genau XML durch RDF genutzt und erweitert wird, zeigt das Beispiel in Abb. 7 der Onlinedokumentationsplattform w3schools. Darin wird im oberen Teil der Grafik eine Tabelle mit Informationen einer CD-Datenbank abgebildet, die im unteren Teil der Abbildung zu einem RDF Dokument überführt wurden. RDF Dokumente beginnen zunächst mit dem in Zeile 3 dargestellten Wurzelknoten *<rdf:RDF>*. In der darauf folgenden Zeile 4 wird durch *xmlns:rdf* ein Namespace spezifiziert, der Elementen mit diesem Prefix eine Namespace-Quelle von w3.org zuweist. Der *xmlns:cd* Namespace ordnet Elementen mit dem cd-Prefix zu den Namespace der Seite „recshop.fake“. Das *<rdf:Description>* Element enthält Beschreibungen des Shopartikels, also der Ressource, die im *rdf:about* Attribut definiert wird.

Title	Artist	Country	Company	Price	Year
Empire Burlesque	Bob Dylan	USA	Columbia	10.90	1985
Hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988

```
1  <?xml version="1.0"?>
2
3  <rdf:RDF
4  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5  xmlns:cd="http://www.recshop.fake/cd#">
6
7  <rdf:Description
8  rdf:about="http://www.recshop.fake/cd/Empire Burlesque">
9    <cd:artist>Bob Dylan</cd:artist>
10   <cd:country>USA</cd:country>
11   <cd:company>Columbia</cd:company>
12   <cd:price>10.90</cd:price>
13   <cd:year>1985</cd:year>
14 </rdf:Description>
15
16 <rdf:Description
17 rdf:about="http://www.recshop.fake/cd/Hide your heart">
18   <cd:artist>Bonnie Tyler</cd:artist>
19   <cd:country>UK</cd:country>
20   <cd:company>CBS Records</cd:company>
21   <cd:price>9.90</cd:price>
22   <cd:year>1988</cd:year>
23 </rdf:Description>
24
25 </rdf:RDF>
```

Abb. 7 Beispiel eines RDF Dokuments (Quelle: w3schools 2016)

Schließlich geben die Kindelemente mit dem vorangestellten „cd“ die Eigenschaften eines konkreten Shopartikels an. Diese Syntax kann zur automatischen Weiterverarbeitung durch RDF-Schema (RDFS) erweitert werden. Mithilfe von RDFS können Zusammenhänge der enthaltenen Beschreibungen, wie Taxonomie- oder Vererbungsstrukturen von Klasse oder Eigenschaften in das Dokument integriert werden. Auf diese Weise kann RDFS in leichtgewichtigen Ontologien zur Domänenbeschreibung und Wissensmodellierung genutzt werden. (KARAKOL 2016, S. 53)

Die durch RDF und RDFS erreichte einfache syntaktische Darstellung von Informationen dient ausdrucksmächtigeren Ontologiesprachen, wie DAML+OIL oder OWL, als Grundlage. Die folgenden Abschnitte erläutern, wie diese Sprachen in der Praxis genutzt werden können.

Web Ontology Language (OWL)

Aufbauend auf RDF und RDFS wird OWL als Nachfolger der Beschreibungssprachen DAML (DARPA Agent Markup Language) und OIL (Ontology Interface Layer) gesehen, welche seit 2001 nicht mehr weiterentwickelt wurden. Seit 2004 dient OWL (Web Ontology Language) als W3C-Standard zur Beschreibung von Ontologien. Damals noch mit Version 1.0, wird OWL 2.0 inzwischen von zahlreichen Systemen unterstützt, darunter bekannte Ontologie Editoren wie „Protege“ oder der „Stanford KSL Ontology Editor“. Durch OWL werden Termini einer Domäne und deren Zusammenhänge in Form einer Ontologie ausgedrückt und als Datenmodell durch XML Syntax gespeichert.

Ziel ist es, Wissen zu strukturieren, es öffentlich zugängig zu machen und online zu verbreiten. OWL ist wesentlicher Bestandteil des Semantic Web, einer Initiative, durch die internetbasierter Content maschinenlesbar aufbereitet wird (CARDOSO UND SHETH 2005). OWL kann dabei mit Abfrage-Sprachen wie SPARQL kombiniert werden, was die Integration in Benutzerschnittstellen vereinfacht. OWL wird in drei Versionen mit unterschiedlicher Ausdrucksstärke unterteilt (KARAKOL 2016):

- *OWL-Lite* zeichnet sich als wenig ausdruckstarke Teilsprache von OWL durch seine einfache Anwendung aus. Es zielt im Wesentlichen auf die Erstellung von Ontologien mit einfachen Taxonomien und unter Einbezug weniger Axiome.
- *OWL-DL* erweitert OWL-Lite um DLs, weshalb sie ausdrucksmächtiger ist und „dem Anwender maximale Beschreibungsfähigkeit zur Verfügung stellt“ (KARA-

KOL 2016). Durch die Kombination aus OWL und DL ermöglicht diese Sprache die Integration von Entscheidungsverfahren, also Algorithmen, um Elemente auf das Vorhandensein bestimmter Eigenschaften zu prüfen.

- *OWL-Full* enthält die beiden Teilsprachen OWL-Lite und OWL-DL und erweitert diese um RDF-Schema. Daher kann diese Sprache als ausdrucksstärkste der drei Teilsprachen verstanden werden.

Schlussfolgerungen basieren in OWL auf der sogenannten Open World Assumption (OWA), bei der grundlegend von einer Existenz von Wissenszusammenhängen ausgegangen wird, solange nicht explizit etwas anderes definiert wurde. Die Basiselemente von OWL sind aus der Objektorientierung bekannt. Es stehen Klassen, Eigenschaften und Instanzen zur Verfügung.

Im Header (Abb. 8, Zeilen 2 bis 4) werden zunächst die erforderlichen Namensräume definiert. An dieser Stelle können zusätzliche Informationen zur Versionskontrolle oder verknüpften, ausgelagerten Ontologien enthalten sein. Durch *rdf:ID* werden Elementen in OWL eindeutige Bezeichner zugewiesen. Hierarchische Beziehungen werden durch Klassen und Unterklassen ausgedrückt: *<owl:Class …>* bzw. *<rdfs:subClassOf …>*. Instanzen werden durch den Namen ihrer zugehörigen Klasse definiert, der gleichzeitig den Namen des XML-Knotens darstellt. In Abb. 8 verdeutlichen die Zeilen 8 bis 11 und die Zeilen 31 bis 33 die Definitionen von Klassen, Unterklassen und Instanzen. (w3c 2016)

Zur Beschreibung von Eigenschaften stehen *ObjectProperty* (hierarchische Objekteigenschaft) und *DataTypeProperty* (Datentypeigenschaft) zur Verfügung. Hierarchischen Objekteigenschaften und Datentypeigenschaften werden durch *domain* einem Definitionsbereich (z.B. in Zeile 23) und mittels *range* ein Wertebereich (z.B. in Zeile 22) zugewiesen. Darüber hinaus lassen sich verschiedene Eigenschaftsmerkmale definieren, beispielsweise transitive, symmetrische, funktionale Eigenschaften oder Umkehrfunktionen, wie Gegenteil-Beziehungen. Eine weitere Besonderheit ist ab Zeile 13 in Abb. 8 dargestellt: Beschränkungen von Eigenschaften, welche zu Ausnahmen innerhalb der Vererbungsstrukturen führen. Diese werden durch *<owl:Restriction>* in der OWL definiert, um beispielsweise Gültigkeitsbereiche einer Eigenschaft anzugeben. In dem gezeigten Beispiel wird auf diese Weise die Konstante *agile* als Wert der Eigenschaft *processType* fixiert. (w3c 2016)

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">

  <owl:Ontology rdf:about=""/>

  <owl:Class rdf:ID="SoftwareDevelopmentProcess"/>
  <owl:Class rdf:ID="ProcessType"/>
  <owl:Class rdf:ID="AgileProcess">
    <rdfs:subClassOf rdf:resource="#SoftwareDevelopmentProcess"/>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#processType"/>
        <owl:hasValue rdf:resource="#agile" rdf:type="#ProcessType"/>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="processType"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:range rdf:resource="#ProcessType"/>
    <rdfs:domain rdf:resource="#SoftwareDevelopmentProcess"/>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:ID="name"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#SoftwareDevelopmentProcess"/>
  </owl:DatatypeProperty>

  <SoftwareDevelopmentProcess rdf:ID="Scrum" name="Scrum">
    <ProcessType rdf:resource="#agile"/>
  </SoftwareDevelopmentProcess>
</rdf:RDF>
```

Abb. 8 Beispiel einer OWL Anwendung

Gleichheitsbeziehungen lassen sich in OWL durch *sameClassAs* für gleiche Klassen, *samePropertyAs* zur Darstellung gleicher Eigenschaften und *sameIndividualAs* für Instanzengleichheit ausdrücken. Durch Schlüsselwörter wie *differentIndividualFrom* können Instanzen mit gleicher Bezeichnung oder Benennung klar voneinander abgegrenzt werden. Hintergrund ist die Zusammenführung von Ontologien unterschiedlicher Autoren, da entsprechende Elemente in unterschiedlichen Ontologien verteilt sein können. Die damit verbundene Identifizierung von Wissensüberlagerung bzw. –differenzierung ist daher notwendiger Bestandteil der OWL. Komplexe Klassenkonstrukte lassen sich durch Mengenoperationen in OWL integrieren. Bekannte Operationen sind beispielsweise *intersectionOf* (Durchschnitt), *unionOf* (Vereinigung), *disjointWith* (Disjunktheit) oder *complementOf* (Komplement). (w3c 2016)

2.3.4 Vorgehensmodelle zur Ontologieentwicklung

Dieser Abschnitt beschreibt Methoden und Vorgehensweisen, um Ontologien neu zu erstellen oder sie in bereits Vorhandene zu integrieren. In der Literatur wird dies unter Ontology Engineering zusammengefasst. Es zielt auf die formale Repräsentation von Konzepten (im Sinne einer Ontologie) und deren Zusammenhänge und bezieht sich dabei auf die bereits beschriebenen Ontologietypen und –sprachen (Abschnitte 2.3.2. und 2.3.3). Bereits Anfang der 1990er Jahre wurden diverse Methoden zur Ontologieerstellung vorgestellt. Einige dieser Vorgehensweisen werden im Folgenden präsentiert. Darunter die Cyc Methode von Lenat und Guha, die Vorgehensweise von GRÜNINGER UND FOX (1995) und die METHONTOLOGY von FERNÁNDEZ-LÓPEZ ET AL. (1997)

Cyc Method von Lenat und Guha (1990)

Die bereits Mitte der 80er Jahre entstandene „Cyc" ist eine große Wissensbasis, die allgemeine Zusammenhänge und Begriffe unsere Weltanschauung beinhaltet. Zur Erstellung der Cyc wurde eine eigens zu diesem Zweck entworfene Sprache verwendet: die CycL (Cyc Language). Sie enthält Inferenzmechanismen, um Vererbungsstrukturen, automatisierte Klassifizierung, Umkehrschlüsse, Bedingungsüberprüfung oder Suchalgorithmen in die Cyc zu integrieren. Darüber hinaus enthält CycL Möglichkeiten zur Überprüfung von Wahrheitsaussagen und zur Detektion von Inkonsistenzen, sowie ein Modul zur Beantwortung allgemeiner Sachverhalte. (GÓMEZ-PÉREZ ET AL. 2004, S. 113)

Das Vorgehen zur Erstellung einer Wissensbasis nach der Cyc Methode kann in drei unabhängige Prozesse unterteilt werden. Beim ersten Prozess wird Wissen manuell aus Quellen, bei denen das Wissen explizit vorliegt, extrahiert. Dieser Prozessschritt ist bei mangelnder Unterstützung von Automatisierungstechniken erforderlich, die zur Überführung von allgemeinem Wissen aus natürlicher Sprache in strukturiertes ontologiebasiertes Wissen hilfreich sind. Hierbei sollen insbesondere drei Zielstellungen erarbeitet werden (ELKAN UND GREINER 1993):

- *Kodierung des zugrundeliegenden Wissens*, das für das inhaltliche Verständnis der Wissensquelle (Internetquelle, Zeitung, Buch, etc.) notwendig ist. Ziel dabei ist, Maschinenlesbarkeit der Wissensbasis zu gewährleisten.

- *Überprüfung von unglaubwürdigen Inhalten*, um herauszufinden, was im Allgemeinen zur Unglaubwürdigkeit führt. Zum Beispiel: Usain Bolt läuft die 100 Meter in weniger als 5 Sekunden.
- *Identifizierung von Fragestellungen*, die Leser nach dem Lesen der Quelle fähig sein sollten, sie zu beantworten.

Im zweiten Prozess von Lenat und Guha dient bereits in der Wissensbasis enthaltenes Wissen zur Kodierung von neuem explizit vorhandenem Wissen. Softwaretools, die zur Analyse von natürlicher Sprache oder zum maschinellen Lernen dienen, leiten allgemeingültiges Wissen aus verschiedenen Quellen auf Basis bereits existierenden Cyc-Wissens ab. Dabei wird menschliches Eingreifen zur Überprüfung und endgültigen Kodifizierung als wesentliches Unterscheidungskriterium zum dritten Prozess gesehen. Dieser ist durch nahezu vollständige Automatisierung während der Überführung von Wissen in die Cyc Ontologie charakterisiert. (ELKAN UND GREINER 1993)

Cyc Ontologien dienen nicht ausschließlich zur Erstellung von Top-Level-Ontologien. Die Cyc Methode schlägt vor, nach Erarbeitung allgemeiner Begriffskonzepte die Repräsentation von Wissen auf spezifische Domänen anzuwenden, um allgemeingültige Begriffe fachspezifisch zu konkretisieren. Nach GÓMEZ-PÉREZ ET AL., S. 115 (2004) wurde diese Methode bis zum Zeitpunkt ihrer Veröffentlichung ausschließlich zur Erstellung der Cyc Wissensbasis genutzt.

Vorgehensweise von GRÜNINGER UND FOX (1995)

Auch GRÜNINGER UND FOX entwickelten bereits 1995 ein Vorgehensmodell zur Entwicklung und Evaluierung von Ontologien. Es entstand aus den Erfahrungen des gemeinsamen Projektes TOVE, welches an der Universität von Toronto durchgeführt wurde, um Wissen zu verschiedenen Unternehmensarchitekturen zu strukturieren.

Abb. 9 zeigt das Vorgehensmodell von GRÜNINGER UND FOX (1995) und seine sechs Prozessschritte zur Erstellung und Evaluierung von Ontologien. Der Prozess beginnt mit dem „Motivating Scenario“. Es umfasst die Erarbeitung von Anwendungsszenarien und Einsatzmöglichkeiten der zu erstellenden Ontologie. Dieser Schritt ist vergleichbar mit der Beschreibung der Vision und Idee bei einem klassischen Softwareentwicklungsprozess. Im zweiten Prozessschritt werden Fragestellungen formuliert, die zur späteren Evaluierung der Ontologie dienen.

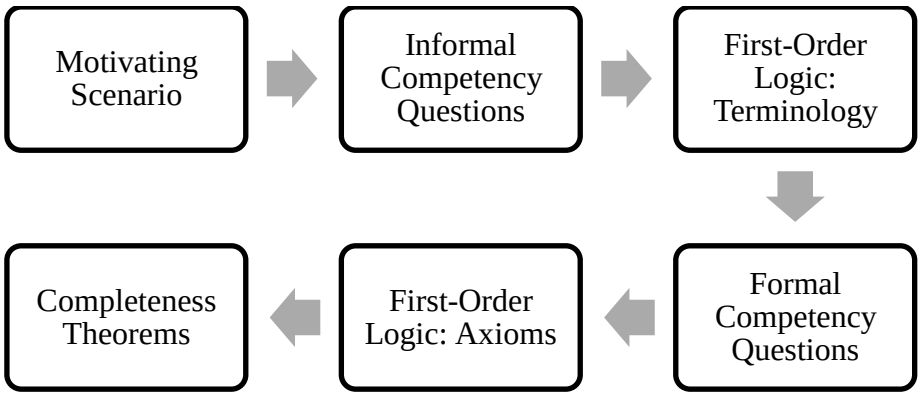


Abb. 9 Erstellung und Evaluierung von Ontologien (nach GRÜNINGER UND FOX 1995)

Die Erarbeitung der sogenannten Informal Competency Questions (deutsch etwa „formlose Fachfragen“) können im Wesentlichen als Testfälle betrachtet werden, die am Prozessende von der Ontologie erfüllt bzw. beantwortet werden müssen. Die Fragen werden in natürlicher Sprache (formlos) formuliert, zum Beispiel:

a. Bei gegebener persönlicher Erfahrung eines Softwareentwicklers und verschiedenen äußeren Einflüssen (ökonomisch bedingt, persönliche oder berufliche Ziele, etc.), welche Software sollte der Softwareentwickler für die Beschreibung von Anforderungen nutzen?
b. Ist Scrum für ein gegebenes Team mit unterschiedlichen Einzelerfahrungen ein geeignetes Vorgehensmodell zur Umsetzung eines bestimmten Softwareprojektes?

Nach GÓMEZ-PÉREZ ET AL. ist eine Ontologie unzureichend umgesetzt, sollten diesen Competency Questions nur „einfachen Abfragen“ entsprechen. Als einfache Abfrage gelten Fragestellungen, die so elementar formuliert sind, dass keine Untergliederung dieser Frage mehr möglich ist. Demzufolge besteht bei Competency Questions die Möglichkeit, sie in kleinere Teilfragen zu verfeinern. Die Beantwortung dieser Teilfragen kann dann genutzt werden, um komplexe Fragestellungen, wie die oben aufgelisteten beantworten zu können. Typischerweise enthalten die Beschreibungen von Competency Questions neben diesen Fragestellungen auch Informationen über die benötigten Eingabedaten, mögliche Bedingungen oder Schranken und Annahmen über Eingabe-Ausgabe-Verhalten.

Der nächste Schritt des Vorgehensmodells (First-Order Logic: Terminology) beschäftigt sich mit der Beschreibung der in die Zielontologie zu integrierenden Terminologien. Diese Beschreibungen erfolgen auf Basis von Prädikatenlogik erster Stufe, um die Integration einheitlich formal vornehmen zu können. Dazu werden die im vorherigen Schritt erstellten

informalen Fragestellungen genutzt, um den Inhalt der Ontologie zu extrahieren. Aus den enthaltenen Terminologien lassen sich Konzepte, Attribute, Axiome und Zusammenhänge ableiten, die mithilfe von Prädikatenlogik formalisiert werden.

Nachdem die Competency Questions durch natürliche Sprache formuliert wurden und die Terminologien in die Elemente der Ontologie überführt werden konnten, erfolgen die Schritte zur Formalisierung des Ontologieinhaltes. In den Schritten vier und fünf werden die Competency Questions und die vorher definierten Axiome in Prädikatenlogik dargestellt. Auf diese Weise kann die Maschinenlesbarkeit der Problem- und Fragestellungen gewährleistet werden. Schließlich wird das Vorgehensmodell von GRÜNINGER UND FOX (1995) beendet, indem formal definiert wird, wann eine von der Ontologie vorgeschlagene Lösung als vollständig erklärt werden kann.

Methontology von FERNÁNDEZ-LÓPEZ ET AL. (1997)

Ein weiteres Modell zur Entwicklung von Ontologien bietet das sogenannte METHONTOLOGY Framework von FERNÁNDEZ-LÓPEZ ET AL. (1997). Es beinhaltet Methodiken zur Verwaltung, Entwicklung und Wartung von Ontologien und dient als Life Cycle Management Methode von Ontologien und deren Prototypen. Der Prozess umfasst im Wesentlichen folgende Prozessschritte (FERNÁNDEZ-LÓPEZ ET AL. 1997):

- *Spezifikationsphase*: In dieser Phase wird ein Dokument erstellt, das insbesondere die Ziele, den Formalisierungsbedarf und den Umfang der zu erstellenden Ontologie enthält.
- *Wissensakquirierung*: Hierbei erfolgt die Extrahierung von Wissen durch manuelle Überführung aus Expertenbefragungen, Literatursichtungen, Abbildungs- und Tabellenanalysen, oder aus bereits existierenden Ontologien. Dabei werden Techniken wie Brainstorming, Interviews, formale und informale Textanalysen oder Wissensgenerierung unter Zuhilfenahme von Softwareunterstützung vorgeschlagen. Das Ergebnis ist ein erstes Glossar mit potenziell relevanten Terminologien und ihren Bedeutungen.
- *Konzeptualisierung*: Ziel dieser Phase ist die Strukturierung des Domänenwissens als Konzeptmodell. Dies erfolgt durch die Erarbeitung eines möglichst vollständigen Glossars aller Terminologien (Glossary of Terms), das aus Klassen, Instanzen, Tätigkeiten und Eigenschaften besteht. Darüber hinaus erfolgt in dieser Phase die

Formalisierung dieser Begriffe, beispielsweise durch eine geeignete Ontologiesprache.

- *Integration*: Zur Beschleunigung der Ontologieentwicklung kann es hilfreich sein, bereits existierende Ontologien auf gleiche oder ähnliche Terminologien zu untersuchen und diese in die geplante Ontologie zu integrieren. Als Ergebnis dieser Phase schlägt METHONTOLOGY ein Integrationsdokument vor, welches die entsprechenden Begriffe, ihre Beschreibungen und die Ontologie ihrer Ursprungsdefinition enthält.
- *Implementierung*: Bei der Implementierung der Ontologie wird eine Umgebung benötigt, die Schnittstellen zu den integrierten Ontologien bereitstellt und ausreichende Unterstützung gängiger Programmiertechniken gewährleistet. Während der Implementierung entsteht Quellcode (auf Basis von CLASSIC, LOOM, Prolog, C#, etc.), der die finale Ontologie verkörpert.
- *Evaluierung*: Um die Korrektheit des enthaltenen Wissens sicherzustellen, erfolgt in dieser Phase die Prüfung des Ausgabeverhaltens der Ontologie (Lösungsvorschläge, Aussagen, etc.) und der entstandenen Softwarelösung.
- *Dokumentation*: Dokumente sind in METHONTOLOGY Bestandteil jeder Phase: In der Spezifikationsphase entsteht ein Anforderungsdokument, während der Wissensakquirierung ein Glossar, nach der Konzeptualisierung ein Konzeptmodell und ein Dokument mit formalisierten Begriffen, nach der Integration ein Integrationsdokument und während der Evaluierung ein Evaluierungsdokument.

Diese sieben Prozessschritte von METHONTOLOGY sind insbesondere bei der Neuentwicklung von Ontologien geeignet, wobei FERNÁNDEZ-LÓPEZ ET AL. (1997) die Integration vorhandener Ontologien als besonders wertvoll einschätzen. Verschiedene Ontologie-Editoren und Frameworks, darunter Pretegé, Apache Jena, SWeDE oder der KSL Ontology Editor bieten diverse Möglichkeiten einzelne Prozessphasen computergestützt durchzuführen.

Weitere Vorgehensmodelle zur Ontologieentwicklung

USCHOLD UND GRUNINGER stellten 1996 eine Methode vor, um Ontologien auf Basis vorgegebener Richtlinien zu erstellen. Ursprünglich dienten diese Vorgehensbeschreibungen zur Erstellung der Enterprise Ontologie, die im Rahmen des Enterprise Projektes an der Universität Edinburgh in Zusammenarbeit mit IBM, Unilever und anderen namenhaften

Unternehmen entstand. Im Wesentlichen unterteilt sich diese Methode in vier Teilprozesse. Der erste Schritt dient der Identifizierung von Wissensquellen und der Herausstellung von Zielstellungen. Die Entwicklung der Ontologie wird im zweiten Teilprozess durchgeführt. Hierbei wird das Wissen zunächst extrahiert, anschließend maschinenlesbar kodifiziert und schließlich gegebenenfalls in existierende Ontologien integriert. Die beiden letzten Teilschritte beschäftigen sich mit der Evaluation des enthaltenen Wissens und mit der Dokumentation der Ontologie. (GÓMEZ-PÉREZ ET AL. 2004, S. 115–119)

Das Projekt „KACTUS“ der Universität Amsterdam wurde mit dem Ziel initialisiert, Wissen aus technischen Domänen wieder zu verwenden. Es untersucht dabei die Bedeutung von Ontologien und schlägt ein Vorgehen für deren Erstellung unter Einbezug von Fremdontologien vor. Der Ontologie-Entwicklungsprozess von KACTUS ist in drei Stufen unterteilt (BREITMAN ET AL. 2007, S. 163). Die Vorstufe bildet eine Spezifikation der Anwendung durch Auflistung anwendungsspezifischer Terminologien. Anschließend erfolgt ein vorläufiger Ontologieentwurf unter Einbezug (Selektion, Erweiterung, Integration) bereits existierender Top-Level Ontologien. Schließlich wird die ausgewählte Ontologie im dritten und letzten Schritt verfeinert, gegebenenfalls neu strukturiert und zur besseren Wartbarkeit modularisiert. (SCHREIBER ET AL. 1995)

Die Semantic Web Best Practices and Deployment (SWBPD) Working Group befasste sich bis 2006 mit der Erarbeitung von Best Practices und Design Patterns zur Erstellung von Ontologien. Der Fokus liegt dabei auf Ontologien, das semantische Web betreffend. Dabei sind Ontologieentwurfsmuster entstanden, die in Teilen oder vollständig als Vorlage für neue Ontologien dienen. NOY UND RECTOR haben die Ergebnisse auf W3C zusammengefasst zu „Defining N-ary Relations on the Semantic Web“. (NOY UND RECTOR 2006)

2.3.5 Inferenzmechanismen beim Ontology Engineering

Bei der Suche nach künstlicher Intelligenz in Ontologien werden von verschiedenen Autoren zwei Stichworte immer wieder erwähnt: Ontology Learning und Reasoner. Dieser Abschnitt gibt einen kurzen Einblick in die Thematiken der automatisierten Wissensgenerierung durch Ontology Learning und dem logischen Schlussfolgern aus existierenden Wissensstrukturen durch OWL Reasoner, wie Racer oder FaCT++.

Ontology Learning

Typischerweise werden Ontologien erstellt, indem eine manuelle Überführung von Wissen aus textbasierten oder grafischen Quellen in eine geordnete Struktur stattfindet. Dieser Prozess beginnt bei den meisten Vorgehensmodellen, wie sie in Abschnitt 2.3.4 beschrieben werden, durch die Sammlung essentieller Begriffe und Tätigkeiten, die in der Zielontologie abgebildet werden sollen. Damit verbunden ist ein enorm hoher Ressourcenbedarf (Zeit, Arbeitskräfte, Kosten, etc.), da die Verfügbarkeit von Wissensquellen durch das Internet in den letzten 20 Jahren stark gestiegen ist. Zur Verringerung des manuellen Aufwandes, liegt der Wunsch nach einer automatisierten Erarbeitung der erforderlichen Wissensartefakte nahe. Diese Automatisierung zur Generierung von Wissen in ontologiebasierten Wissensbasen wird als Ontology Learning bezeichnet.

Ontologien können durch ihre verschiedenen Abstraktionsebenen charakterisiert werden. Sie sind in Abb. 10 schematisiert und bilden die Grundlage für Ontology Learning (BUITELAAR ET AL. 2005).

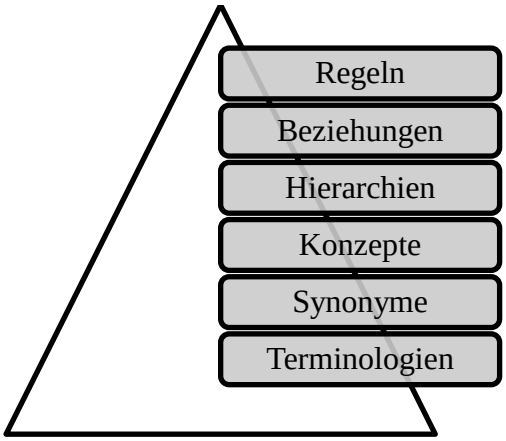


Abb. 10 Ebenen des Ontology Learning (nach BUITELAAR ET AL. 2005, S. 5)

Das Fundament des Ontology Learnings bildet die Suche nach passenden Terminologien (Substantive, Verben, Adjektive, etc.). Ansätze zur Realisierung dieser Aufgabe finden sich vor allem im Information Retrieval. Anschließend müssen Begriffe herauskristallisiert werden, die zwar syntaktische Unterschiede aufweisen, aber dieselbe Bedeutung verkörpern (Synonyme). Dazu werden Techniken verwendet, bei denen durch Analyse semantischer Verknüpfungen (beispielsweise über das Internet) Rückschlüsse auf die Bedeutung gezogen werden können. Auf Konzept-Ebene werden die gefundenen Begrifflichkeiten durch Definitionen, lexikalische Verweise oder andere inhaltliche Ergänzungen genauer

beschrieben. Dafür werden Ansätze wie das *Latent Semantic Indexing* oder verschiedene Werkzeuge wie *Know-it-all* oder *OntoLearn* genutzt. Diese Techniken und Softwaretools nutzen mathematische Operationen zur Mustererkennung von Begriffen und Zusammenhängen in Texten. (BUITELAAR ET AL. 2005)

Um Vererbungsstrukturen zwischen den Konzepten in die Zielontologie zu integrieren, werden auf der Hierarchie-Ebene Taxonomien abgeleitet. Zur Bewältigung dieser Aufgabe dienen Methoden wie das Erkennen von syntaktischen Gleichheiten (lexicosyntactic patterns) oder Cluster-Algorithmen, um automatisch hierarchische Beziehungen zwischen den Konzepten zu erkennen. Beziehungen, Eigenschaften, Zuordnungen oder Reichweiten von Konzepten werden beispielsweise durch Assoziationsregeln gelernt. Ansätze, um Beziehungen in Ontologien aus Texten zu lernen, sind im Vergleich zu den Techniken der unteren Ebenen weniger zu finden. Den Abschluss des Ontology Learning Prozesses bildet die automatische Ableitung von Axiomen. Dabei muss zunächst festgestellt werden, welche Konzepte und Assoziationen Axiome benötigen und wie genau diese aufgebaut sein müssen. Je nach Prozesstiefe können auf Basis dieser Axiome komplexere Zusammenhänge zwischen den Konzepten und Beziehungen abgeleitet werden. (BUITELAAR ET AL. 2005)

Es existieren bereits verschiedene Software-Tools zur Anwendung des Ontology Learning. *Text2Onto* der Universität Karlsruhe, welches von CIMIANO UND VÖLKER (2005) vorgestellt wurde, ist beispielsweise ein Vertreter zur strukturierten Überführung von Wissen aus textbasierten Quellen. Weitere Tools zur Textanalyse sind *OntoLancs*, *DINO* oder *OntoGen*. Bedeutende Softwarewerkzeuge zur logischen Schlussfolgerung beim Ontology Learning sind *OntoComp* (Universität Dresden), *RELExO* (Universität Karlsruhe) oder der *DL-Learner* (Universität Leipzig). Als Ontology Learning Tools, die sowohl textbasierte Wissensakquise, als auch logische Schlussfolgerung zur Wissensüberführung nutzen, sind *LeDA* und *SOFIE* der Universität Karlsruhe bzw. Saarbrücken zu erwähnen. (CIMIANO 2006, S. 23–26)

Herausforderungen beim Ontology Learning bestehen in erster Linie in Bezug auf die korrekte und konsistente Integration von Wissen. Daher wird im Zusammenhang mit Ontology Learning meist von einer Teilautomatisierung gesprochen, bei der menschliches Eingreifen zur endgültigen Überprüfung der Wissensintegration notwendig ist.

Reasoner

Wie bereits erwähnt kann OWL in drei Ausprägungen untergliedert werden: OWL-Lite, OWL-DL und OWL-Full. Die genaue Beschreibung der einzelnen Teilsprachen kann Abschnitt 2.3.3 entnommen werden. Gegenüber OWL-Lite ist einer der Vorteile von OWL-DL und OWL-Full die Verarbeitbarkeit durch einen sogenannten Reasoner (deutsch etwa Schlussfolgerer). Möglich wird dies, da OWL-DL und OWL-Full Beschreibungslogiken enthalten, die das integrierte Wissen formalisieren. Um Reasoner in die OWL-basierte Ontologieerstellung einzubeziehen, stehen verschiedene Werkzeuge zur Verfügung, wie Racer, FaCT++ oder HermiT. Einbinden lassen sich diese Werkzeuge durch die „OWL-Reasoner"-Schnittstelle.

Reasoner können beispielsweise feststellen, ob eine bestimmte Klasse die Unterklasse einer anderen Klasse darstellt. Wird dieses Verhalten auf alle in der Ontologie enthaltenen Klassen angewandt, können Reasoner auf Taxonomien bzw. Klassenhierarchien innerhalb der Ontologie automatisch schließen. Weiterhin dienen Reasoner in Ontologien dazu, die Konsistenz von enthaltenem oder neuem Wissen zu überprüfen. Auf Basis der Axiome und Bedingungen, die Klassen und Beziehungen von Ontologien enthalten können, kann ein Reasoner Instanzen dieser Elemente logisch prüfen. Eine Klasse gilt als inkonsistent, wenn es aufgrund ihrer Axiome nicht zu einer Instanziierung kommen kann. Darüber hinaus dienen Reasoner als Werkzeug, um semantische Gleichheit von Konzepten zu erkennen (Synonyme bzw. Äquivalente finden) und als Logikinterpreter, um Abfragen in Ontologien zu ermöglichen. Inferenzprobleme, die mithilfe von Reasoner gelöst werden können sind:

- Transitivitätseigenschaften: Existieren Beziehungen zwischen zwei nicht assoziierten Objekten auf Basis von Transitivität?
- Disjunktheit von Klassen erkennen: Wenn eine Instanz einer bestimmten Klasse angehört, kann sie eventuell keine Instanz einer bestimmten anderen Klasse sein.
- Klassenzugehörigkeit: Ist eine Klasse in einer anderen Klasse enthalten?
- Instanzerzeugung (Retrieval): Finde alle Instanzen einer bestimmten Klasse oder mit einer bestimmten Eigenschaft.

Das Ziel dabei bildet die Ableitung von Wissen auf Basis der in der Ontologie enthaltenen Wissenszusammenhänge. Reasoner werden hauptsächlich auf Beschreibungslogik und nicht auf Prädikatenlogik erster Stufe (First Order Logic) angewandt, da First Order Logics

nur formalisierte Wissenszusammenhänge kennt und Probleme daher teilweise unvollständig aufgelöst werden. (HORRIDGE ET AL. 2004)

Einer der meist genutzten Reasoner ist Racer. Er wurde an den Universitäten Lübeck und Concordia (Kanada) entwickelt und ist für zahlreiche OWL-Entwicklungswerkzeuge, wie Protégé verfügbar. Racer steht für Renamed ABox and Concept Expression Reasoner. Er kann als Bibliothek in Anwendungssoftware oder Webprojekten eingebunden oder per API-Aufruf verwendet werden. Zahlreiche Dokumente, darunter umfangreiche Dokumentationen, Anwendungsbeispiele und OpenSource-Quellen stehen Anwendern auf den Internetseiten der Universität Lübeck zur Verfügung. (HAARSLEV 2016)

Grenzen und Herausforderungen

Eine der wesentlichen Herausforderungen auf dem Gebiet des Ontology Engineering ist die Bewertung der Qualität von ontologiebasiertem Wissen. Da eine Ontologie eine isomorphe Repräsentation von Begriffen ist, die unsere Realität beschreiben, ist es problemlos möglich, zusammenhangsloses oder inkonsistentes Wissen zu integrieren (JANSEN ET AL. 2008, S. 38). Es wird versucht, durch verschiedene Inferenzalgorithmen von Reasonern dieser Inkonsistenz entgegen zu wirken. Jedoch können keine Fehler vermieden werden, die bereits während der Modellierung des Wissens, beispielsweise bei der händischen Erstellung von Axiomen oder Bedingungen, entstehen. Weiterhin kann die Plausibilität von Begriffen nur bedingt gewährleistet werden (JANSEN ET AL. 2008, S. 40–42). So ist es bei der Integration von Mustererkennungstechniken, zum Beispiel durch Verknüpfung mit künstlichen neuronalen Strukturen, bedeutungslos, ob die Begriffe der Realität entsprechen oder nicht. Demnach können Aussagen über den Wahrheitsgehalt einer Schlussfolgerung kaum oder nur mit hohem Aufwand getroffen werden.

In der Praxis werden Ontologien meist als schlichter Wissensspeicher verwendet, was zu einem geringen Formalisierungsgrad innerhalb der Ontologie führen kann. Damit verbunden ist die fehlende Möglichkeit Wissen inferenzbasiert zu speichern, sodass sich häufig keine Interpretierbarkeit der gespeicherten Wissensstrukturen durch Lern- und Schlussfolgerungsmethoden bieten. Zudem bilden zahlreiche Anwendungen lediglich Taxonomien und Klassenhierarchien ab. In diesem Zusammenhang wird auf die Integration von Axiomen oder Bedingungen und damit auf den vollen Funktionsumfang von Beschreibungslogik verzichtet.

Das Gebiet des Ontology Engineering umfasst mit all seinen Facetten ein breites Anwendungsfeld, was zu einer Vielzahl an Werkzeugen, Vorgehensmodellen, Ontologiesprachen, Reasonern oder Methoden des Ontology Learnings führt. Viele dieser Artefakte sind auf den Einsatz in verschiedenen Domänen ausgerichtet. Einerseits wird ihre Anwendung auf diese Weise vereinheitlicht, andererseits durch den Versuch, multiple Anwendungsfälle abzudecken, stark verkompliziert. Darüber hinaus basieren die wenigsten Artefakte auf der Überführung von Wissen aus natürlicher Sprache in ein stark formalisiertes Datenmodell. Zwar kann Wissen mithilfe des Ontology Learnings aus natürlicher Sprache in Ontologien akquiriert werden, die Formalisierung des Wissens basiert dann jedoch auf komplizierter Beschreibungslogik oder Prädikatenlogik erster Ordnung. Eine objektorientierte Betrachtungsweise ontologiebasierter Wissensstrukturen kann diesbezüglich eine neue Sicht auf Ontologien eröffnen und diese stark vereinfachen.

3 Modellbeschreibung

3.1 Ziele des Modells

Insgesamt lässt sich nach der Literaturrecherche festhalten, dass keine der identifizierten Literaturquellen alle Anforderungen an den gesuchten Lösungsansatz erfüllt und nur wenige Arbeiten dem hier vorgestellten Thema sehr nahe kommen. Einige Anforderungen werden in Teilen von verschiedenen Publikationen abgedeckt. Darunter BEYDOUN ET AL. (2014), durch die beispielsweise eine Ontologie vorgestellt wird, die Anforderungen entgegen nimmt, zu Wissensartefakten verarbeitet und während der Softwareentwicklung einen Konsistenzcheck vornehmen kann. Die Autoren beziehen sich jedoch nicht auf Deduktionsalgorithmen oder eine Unterscheidung von beschreibender und konkreter Ebene. Letztere ist in der Methode von GUIZZARDI UND ZAMBORLINI (2014) zur Modellierung von Ontologien zwar integriert, jedoch sind Datenerfassungen und soziale Einflüsse nicht explizit modellierbar. Zudem enthält ihre Modellierungsmethode keine konkreten Algorithmen für automatisierte Schlussfolgerungen aus dem enthaltenen Wissen. Günther Ruhe beschäftigte sich bereits in zahlreichen Publikationen mit der Thematik „Software Engineering Decision Support“. Seine Veröffentlichung RUHE 2003 enthält ausführliche Erläuterungen zur Bedeutung von entscheidungsunterstützenden Systemen im Software Engineering. Er beschreibt jedoch nur auf konzeptioneller Basis, wie Wissen die Softwareentwicklung entlang ihrer einzelnen Teilgebiete unterstützen kann.

Um Ontologien zu erstellen, existieren verschiedene Vorgehensweisen (Abschnitt 2.3.4, Seite 31), welche die Ontologieerstellung als Prozess betrachten, der bei der Akquirierung von Wissen beginnt und bei der Evaluierung der fertigen Ontologie endet. Das Ziel des Modells dieser Arbeit ist nicht einen derartigen Vorgehensprozess zu beschreiben. Vielmehr soll eine Möglichkeit geschaffen werden, Wissen aus der Domäne des Software-Engineering in einer formalisierten Form zu strukturieren und darstellbar zu machen. Es

handelt sich daher um ein abstraktes Modell, das die Architektur für die Anfertigung einer speziellen Ontologie verkörpert.

Dieses Modell soll neben der Ontologie-Architektur auch Methoden bereitstellen, um Rückschlüsse aus dem in der Ontologie enthaltenem Wissen ableiten zu können. Schlussfolgerungen dieser Art sollen Antworten auf folgende, exemplarisch aufgelistete Fragestellungen darbieten können:

- Ist Scrum als Softwareentwicklungsmethode für ein bestimmtes Projektteam geeignet, oder nicht?
- Handelt es sich bei dem vorliegendem Codeausschnitt um ein Architekturmuster?
- Um welches Architekturmuster handelt es sich im vorliegenden Fall?
- Woraus besteht Software Engineering?
- Welche Aufgaben sind mit Software Engineering verbunden?
- Welche Eigenschaften hat ein „guter Programmierer“?
- Welche Software existiert, um Anforderungen zu dokumentieren?
- Wozu dient Jira? Wer oder was kann Jira nutzen?
- Wer oder was kann Anforderungen beschreiben?
- …

Diese Fragestellungen stellen beispielhaft dar, welche Zielstellungen mit dem hier vorgestellten Modell zur Ontologieerstellung verfolgt werden. Besonders Ziel stellt die Minimierung der Modellierungskomplexität dar. Ein Modellierer, der nach diesem Modell Ontologien erstellt, soll unabhängig von komplexen Ontologiesprachen wie OWL-DL oder OWL-Full agieren können.

3.2 Ebenen und Elementklassen

Das Modell kann als Fünf-Ebenen-Architektur beschrieben werden. Die Ebenen ergeben sich aus einer Eingabe-, einer Datenüberführungs-, einer Informationsverarbeitungs- und einer Abstraktionsschicht. Die Ebenen repräsentieren unterschiedliche Informationsgrade, wodurch Wissen in verschiedener Detailtiefe gespeichert und verarbeitet werden kann (Abb. 11). Jede Ebene enthält mindestens ein Element. Elemente sind durch sich wiederholende geometrische Figuren in Abb. 11 dargestellt und können durch verschiedene Assoziationstypen miteinander in Beziehung gebracht werden. Diese Relationen können zwischen

Elementen einer Ebene oder zwischen Elementen verschiedener Ebenen gebildet werden. Assoziationsregeln, wie sie beispielsweise auf dem Gebiet der Künstlichen Intelligenz oder dem Data Mining angewandt werden, dienen dabei als geeignetes Mittel maschinelles Lernen in das Modell zu integrieren (McNicholas und Zhao 2009).

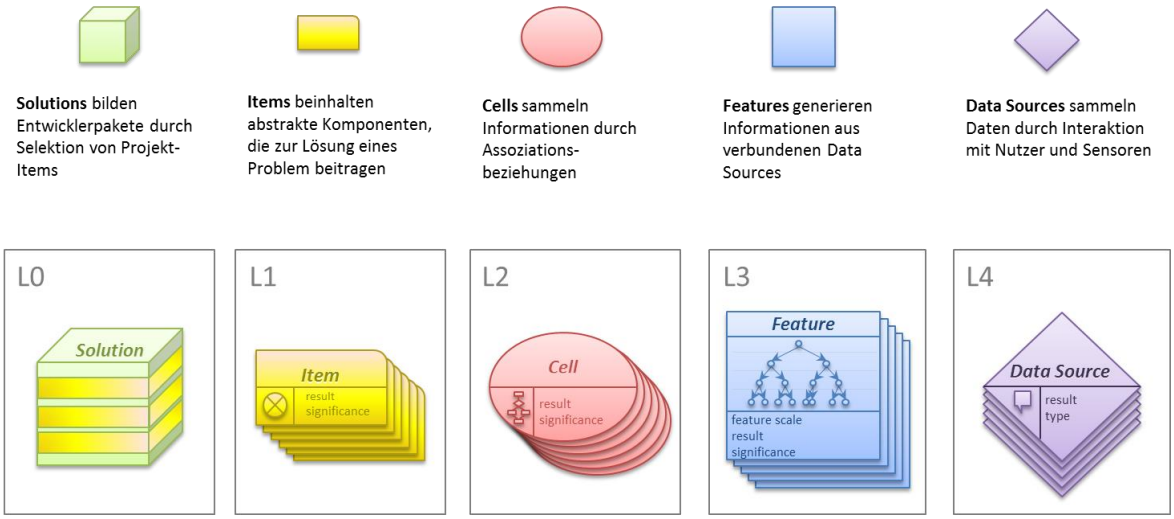


Abb. 11 Darstellung der unterschiedlichen Ebenen der Ontologie

Das Prinzip der losen Kopplung wird auf jeder Ebene des Models angewandt, was zu einer hohen Austauschbarkeit und Erweiterbarkeit aller enthaltenen Elemente führt (Zhu 2005, S. 156–157). Jedes Element enthält implizit eine Problemstellung und explizit ein Ergebnis, welches durch eine elementspezifische Logik gefunden wird und von anderen Elementen genutzt werden kann. Die hierarchische Struktur des Modells dient zusammen mit der Verknüpfung von elementbezogenen Logikkomponenten dazu, übergeordnete Problemstellungen auf kleinere verarbeitbare Probleme aufzuteilen. Diese Vorgehensweise ist eines der Hauptparadigmen der Künstlichen Intelligenz (Norvig 1992; Bonabeau et al. 1999; Brooks 1986).

Die oberste Ebene *L*0 bildet die *Solution*-Ebene. Das Ziel dieser Schicht ist die Erzeugung von Entwicklerpaketen, die aus unterschiedlichen Elementen zur Umsetzung des Projekts zusammengesetzt sind. Die Entscheidungen, welche Elemente miteinander kombiniert werden, basiert auf einem einfachen Tauglichkeitstest. Dabei überprüfen alle Assoziationen der Items ihre Projekttauglichkeit. Daraufhin bestimmt jedes Item die verknüpfte Zelle mit dem höchsten Tauglichkeitswert. Die entsprechenden Ergebnisse werden zu einer projektspezifischen Solution abstrahiert. Dieser Prozess wird als Deduktion bezeichnet und in

Kapitel 1 ab Seite 71 genauer erläutert. Die weiteren Ebenen ($L1$ - $L4$) werden in den Abschnitten 3.2.2 und 3.2.3 näher erläutert.

3.2.1 Betrachtung des abstrakten Modells als Graph

Abb. 12 zeigt das Metamodell als Graph, bei dem die Elemente als Knoten und die Assoziationen als Kanten betrachtet werden können. Dabei ist eine baumartige Struktur des Modells zu erkennen, bei dem ein Solution-Element den Wurzelknoten bildet. In der Grafik wird jedoch auch deutlich, dass Kreisbeziehungen zwischen verschiedenen Elementen auftreten können. Diese sind mit einfachen Kompositionsmustern (baumartigen Architekturen) nicht performant abbildbar, da eine Komposition sich selbst als Kindkomposition ihres eigenen Kindknoten übergeben bekäme. Darüber hinaus muss sowohl bei einer Deduktion als auch bei einer induktiven Inferenz ein Einstieg an jedem beliebigen Element gewährleistet sein. Ausgehend von genau diesem Einstiegselement werden die Pfade gesucht, die zur Schlussfolgerung führen. Innerhalb des Modells kann demnach jedes beliebige Element ein Wurzelknoten bilden, was die Betrachtung als einfachen Baum irrelevant werden lässt.

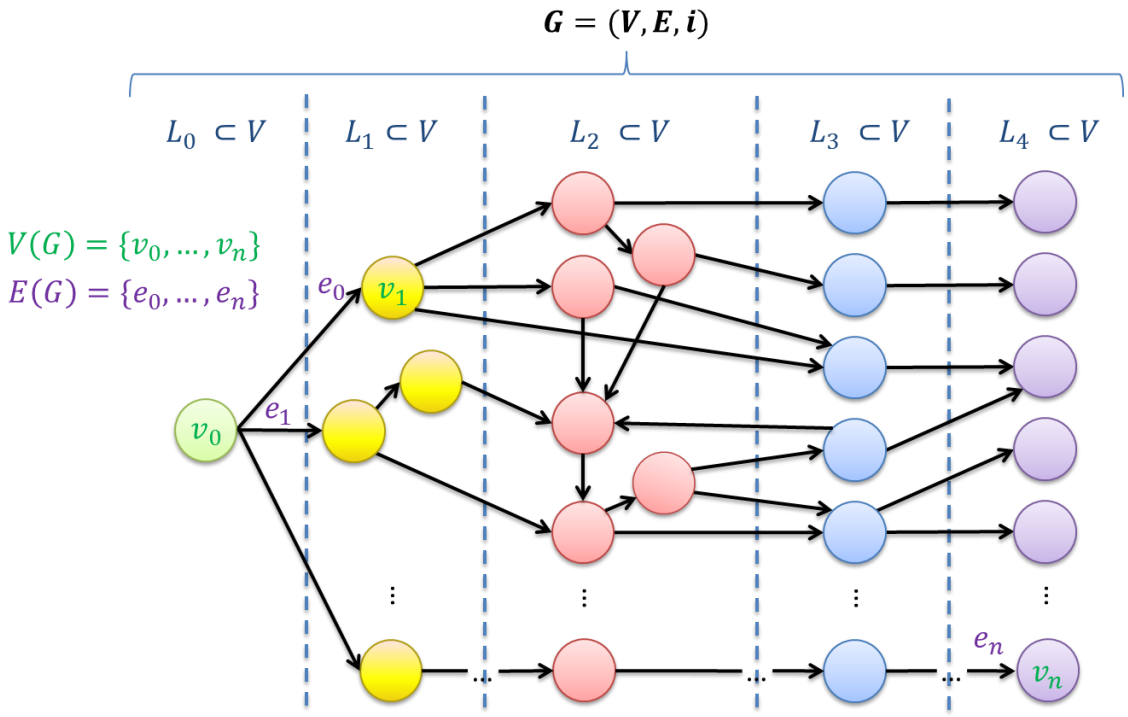


Abb. 12 Darstellung des Graphen

Werden die fünf Schichten mithilfe mathematischer Definitionen aus Graphen- und Mengentheorie betrachtet, handelt es sich bei dem Modell um einen gerichteten, kantengewichteten Graphen $G = (V, E, i)$. (GOLUMBIC 2004; FOREMAN UND KANAMORI 2010)

Die Ebenen ($L0$ bis $L4$) bilden geordnete Paare (V, E), dabei ist

- $V(G)$ die endliche Menge aller Knoten (Elemente),
- $E(G)$ die Menge aller Kanten (Assoziationen) und
- $i = i_G$ die Abbildung, die jeder Kante $e \in E$ ein Paar $i(e) = \{x, y\}$ von Elementen $x, y \in V$ zuordnet.

Wie Abb. 12 zu entnehmen ist, kann die Menge V in fünf Teilmengen untergliedert werden:

$$L_0, L_1, L_2, L_3, L_4 \subset V(G)$$

Sie enthalten die Elemente der unterschiedlichen Ebenen des Models. Die Ebenen der Cells und Items stellen dabei eine Besonderheit gegenüber den übrigen Ebenen dar. Sie können als sinnvolle Teilgraphen $C_1, C_2 \subset G$ betrachtet werden. Auf diesen Ebenen sind, im Gegensatz zu den anderen Schichten, Verknüpfungen (also Kanten) zwischen den Knoten (innerhalb der Ebene) möglich. Somit sind in C_1 und C_2 jeweils maximal

$$\binom{n}{2} = \frac{n(n-1)}{2} \; mit \; n = |L_2| \text{ bzw.}$$

$$\binom{m}{2} = \frac{m(m-1)}{2} \; mit \; m = |L_1|$$

Assoziationen vorhanden (BISWAL 2015, S. 101). Diese maximal mögliche Anzahl an Knoten von L_2 und L_1 impliziert jedoch die Vollständigkeit der jeweiligen Teilgraphen. Sie kann nur erreicht werden, indem jedes Element einer Ebene mit jedem anderen Element derselben Ebene verknüpft ist.

Durch die Implementierung von Kantengewichten kann der Deduktionsprozess beschleunigt werden. Ziel ist das Durchlaufen des Graphen auf priorisierten Pfaden, um schneller an bedeutsame Elemente (Knoten, Informationen) zu gelangen. Dabei wird ähnlich dem PageRank-Algorithmus von der Zentralität auf die Bedeutung eines Elements geschlossen. Die Bedeutung eines Knotens in einem Netz wird demnach in Abhängigkeit der Anzahl

seiner Verbindungen gestellt (SAINT-AUBIN UND ROUSSEAU 2008, S. 269). Genaue Erläuterung zur Gewichtung erfolgen in Abschnitt 4.2.

3.2.2 Features, Data Sources und ihr Zusammenhang

Die Ebenen *L*3 und *L*4 enthalten die *Feature*- und *Data Source*-Elemente des Models. Sie dienen als Anwendungsschicht und bilden die zentrale Kommunikation mit einzelnen Projektbeteiligten, den Nutzern. *L*4 erfasst durch ein einfaches Request/Response-Prinzip elementare Sachverhalte und ermöglicht auf diese Weise einen Istbestand des Projektumfeldes aufzunehmen. Data Source-Elemente dienen also ausschließlich dem Zweck Daten in das Modell zu integrieren. Diese können verschiedenen Ursprungs sein. Sie können aus Nutzerfragen, Messinstrumenten oder Sensoren als Eingabeinstrumente in das System erfolgen.

Jedes Feature bildet dabei eine Einflussgröße auf das Projekt. Dabei wird jedem Feature ein Element aus der Data Source-Ebene zugeordnet. Beispiele für Feature sind „gesamtes Projektbudget“, „verwendetes Betriebssystem“, „persönliche Motivation“ oder „persönliche Erfahrung mit einem Prozessmodell“. Da Anzahl und Einfluss dieser Feature eng mit technologischen Fortschritten und neuen Erkenntnissen aus Wissenschaft und Forschung korrelieren, unterliegen sie einem besonders hohen Grad an Veränderbarkeit.

Um eine Analyse der Ergebnisse zu ermöglichen, werden Feature in Nominal-, Ordinal- und Metrische-Skalen klassifiziert. Diese Klassifizierung erfolgt in Abhängigkeit der verknüpften Datenquelle und unter Berücksichtigung des erwarteten Ergebnisses. Die Klassifizierung der Feature, die mit Fragen verknüpft sind, richtet sich nach den Fragetypen „Multiple-Choice“ und „Single-Choice“, bzw. nach unterschiedlichen „Input“-Datentypen, wie Integer oder String. „Input“-Datentypen sind typischerweise Messinstrumente oder Fragen bei denen eine Nutzereingabe erforderlich wird. Multiple-Choice und Single-Choice-Feature sind prinzipiell mit einer Nutzerfrage verbunden. Bei Multiple-Choice Fragen kann der Nutzer aus verschiedenen Antwortmöglichkeiten mehrere auswählen. Bei Single-Choice Fragen hingegen wählt der Nutzer nur genau eine Antwort aus den vorgegebenen Antwortmöglichkeiten aus. Die Auswahlmöglichkeiten ergeben sich aus Verknüpfungen zu Cells. Die Ergebnisse der verbundenen Cells werden vom Feature in Abhängigkeit der Nutzerauswahl als „zutreffend“ (true) oder „nicht zutreffend“ (false) geschalten.

Abb. 13 visualisiert den Zusammenhang und die Klassifizierungsstufen der beiden Ebenen Data Source und Feature.

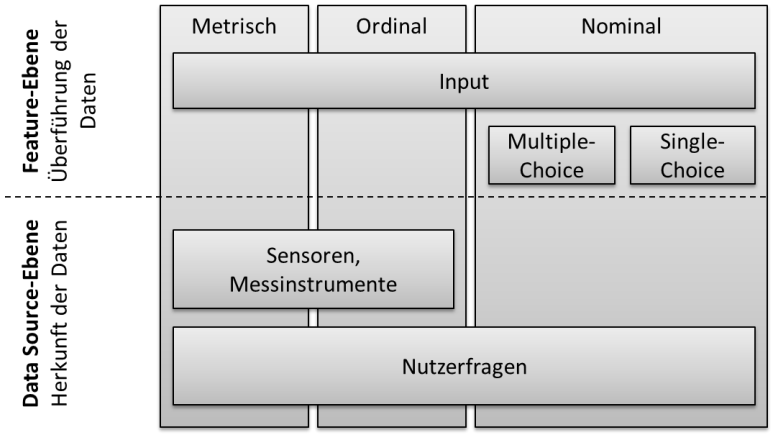


Abb. 13 Klassifizierung von Daten in Feature- und Data Source-Ebene

3.2.3 Cells und Items als A-Box und T-Box

Jedes Feature ist mit mindestens einer *Cell* (oder einem *Item*) verbunden. Die Gesamtheit aller Cells wird auf einer weiteren Schicht des Models erfasst, der Ebene $L2$. Nach HASENKAMP UND ROßBACH könnten Cells als Semantikbausteine und nach LANDAUER 1998 als Kontextblöcke verstanden werden. Sie sammeln durch Auswertung ihrer Assoziationsbeziehungen Daten und interpretieren diese durch hinterlegte Algorithmen zu Informationen. Damit repräsentieren sie die Grundlage zum Erzeugen von Wissen. Durch Verknüpfung untereinander können sie, neben gezieltem Auswerten benachbarter Feature, auch auf Informationen anderer Cells zugreifen und in die eigene Analyse einbeziehen. Cells sind nach Anzahl ihrer Assoziationsbeziehungen unterschiedlich gewichtet und können Abhängigkeiten untereinander aufweisen. Eine auf diese Weise aufgebaute Abhängigkeit ist zum Beispiel *Scrumwise* (ein softwaregestütztes Werkzeug zur Softwareentwicklung nach Scrum) und *Scrum* als Entwicklungsprozess.

Die in den Cells gespeicherten Informationen werden von sogenannten *Items* genutzt, welche in ihrer Gesamtheit die Ebene $L1$ bilden. Jedes Item beinhaltet eine abstrakte Komponente, die für die Umsetzung eines spezifischen Projektes erforderlich sein kann. Items dienen daher als partielle Lösung, um auf eine bestimmte Problemstellung oder Anforderung zu reagieren. Daher müssen Items besonders genau untersucht werden, inwiefern sich ihr Einbezug auf den Projekterfolg auswirken würde. Sie können in unterschiedlichen Pro-

jektabschnitten und dabei zur Bewältigung verschiedener Aufgaben von Nutzen sein. Beispielsweise kann „Requirements Engineering“ als Item symbolisiert werden. Dieses Item ist wesentlicher Bestandteil des Produktmanagements und zwingend erforderlich für einen erfolgreichen Abschluss des Projektes. Weitere Beispiele für Items sind „Software Development Process“ oder „Software Architecture Pattern“.

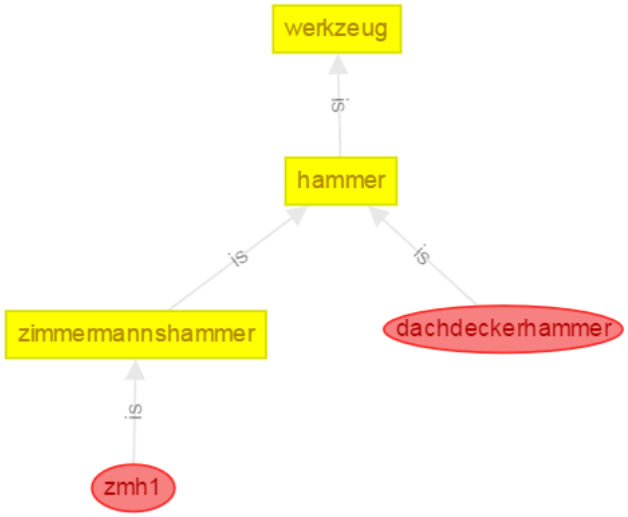


Abb. 14 Einfaches Beispiel: Cells als Blätter eines "is“-Pfad-Baumes

Zum besseren Verständnis, worin sich Items und Cells unterscheiden: $L1$ (Items-Ebene) dient als beschreibende Sichtweise (T-Box, terminological component) auf das im System gespeicherte Wissen, während $L2$ (Cells-Ebene) eine entscheidende Sichtweise mit konkreten Instanzen (A-Box, assertion component) liefert. Cells sind demnach grundsätzlich etwas Konkretes, wohingegen Items etwas Allgemeines, Abstraktes oder Klassifizierendes darstellen.

Neben der manuellen Modellierung von T-Box und A-Box kann auch ein automatisiertes Schlussfolgern zur Unterscheidung der beiden Elementklassen erfolgen. Eine wesentliche Aufgabe von Ontology Reasoner ist das Finden von Unterklassen (vgl. Abschnitt 2.3.5). Wird dieses Vorgehen auf einen is-Pfad-Baum angewandt (Erläuterungen zur is-Assoziation erfolgen in Abschnitt 3.3.2), kann jedes Blatt des Baumes als Konkretisierung seines Elternknoten verstanden werden. Da alles Konkrete im Modell als Cell interpretiert wird, bilden die Enden dieses Baumes Cells und alle inneren Knoten Items. Abb. 14 versucht diesen Zusammenhang zu visualisieren.

3.2.4 Activities und Combinings

Um Aufgaben, Funktionen, Tätigkeiten oder Aktivitäten in die Wissensbasis einzupflegen, stellt das Modell sogenannte Activities zur Verfügung. Sie dienen in erster Linie der Integration ausführbarer Handlungen, die eine Rolle, ein Individuum oder ein Subjekt durchführen kann. Im deutschen und englischen Sprachgebrauch sind Activities meist Tätigkeitswörter (Verben), die einem aktiven oder einem passiven Charakter entsprechen können. Abb. 15 enthält dazu zwei Beispiele.

Abb. 15 Aktive und Passive Activities

Ein Mensch kann laufen. Dieser Sachverhalt ist eindeutig und benötigt keine weiteren Informationen, um das zugrunde liegende Wissen nutzen zu können.

Jira als Software kann etwas organisieren. Das Schlüsselwort zur Unterscheidung von aktiv und passiv ist in diesem Fall das Wort „etwas“, das implizit nach einer zusätzlichen Information verlangt: Was genau kann Jira organisieren? Ohne Zusatzinformationen würde passives Wissen keine ausreichende Grundlage zur Interpretation bieten.

Zur Eingliederung in die mathematische Beschreibung des Modells sind Aktivities definiert als Elemente der Menge $A \subset L_2$. Bezogen auf das Beispiel in Abb. 15 gilt demnach: $walk, organize \in A \subset V$.

Activities verkörpern den Eingriff einer manuellen Handlung, bei der ein Verhaltensmuster implizit beim Menschen vorliegt. Sie definieren also nicht die Inferenzpfade und können daher selbst Gegenstand der Inferenz sein. Darüber hinaus können sie auch mit expliziten Verhaltensmustern versehen werden, um beispielsweise bestimmte Funktionen automatisiert durchführen zu lassen. Diese Verhaltensmuster können Schaltungsabläufe, Geschäftsprozesse, Algorithmen, Bewegungsabläufe oder die Initialisierung von Mess- und Sensortechniken enthalten.

Die Entscheidung, ob eine Tätigkeit als aktiv oder passiv einzustufen ist, ist stark von der Wahrnehmung des Modellierers abhängig (mentales Modell). *Reden* (aktiv) ist bei jedem

Menschen eindeutig mit einem bestimmten Verhaltensmuster verbunden, was dieser Tätigkeit wenig Interpretationsspielraum bietet. *Beschreiben* (passiv) hingegen kann je nach Subjekt mit unterschiedlichem Verhalten verbunden sein. Eine Zusatzinformation *Anforderung beschreiben* ist dagegen mit einem (weitestgehend) einheitlichen mentalen Modell verbunden. Daher führt die Verknüpfung eines Objektes[2] mit einer passiven Aktivität meist zu einer Informationsanreicherung. Zu diesem Zweck stellt das Modell Combinings bereit. Ein Combining ist eine Zusammensetzung zweier Elemente: Zum Beispiel die eben erwähnte Verknüpfung von passiver Aktivität mit einem Objekt. Möglich ist auch die Verknüpfung einer Eigenschaft mit einem Subjekt oder die Verbindung von Eigenschaften und Objekten. Combinings sind Elemente der Menge $C \subset L_2$.

Abb. 16 zeigt anhand einer Erweiterung des vorherigen Beispiels die Verwendung von Combinings. Im Beispiel sind neben zwei Items (software, requirement) und einer Cell (jira) auch eine Activity (organize) und ein Combining (organize requirement) zu sehen. Die genaue Beschreibung der Verknüpfungstypen „is“, „can“ und „part-of“ erfolgt in Abschnitt 3.3.2.

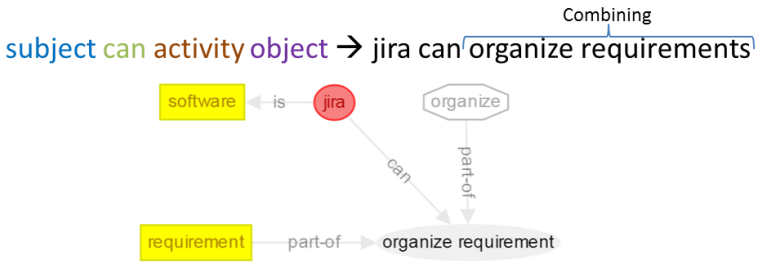


Abb. 16 Combining mit passiver Activity

Würde die Abbildung in natürlicher Sprache formuliert werden, entspräche dies folgendem Wortlaut: „Jira ist eine Software. Jira kann Anforderungen organisieren.“ Die Umkehrung der Herangehensweise, also die Überführung von Wissen aus natürlicher Sprache in das Modell (Ontology Learning, Abschnitt 2.3.5), lässt folgende Schlussfolgerung zu:

[2] Objekte und Subjekte sind aus Sicht des Modells Instanzen, also Cells. Sie unterscheiden sich durch ihre Fähigkeit der Wahrnehmung. Während Subjekte das Erkennende darstellen, beschreiben Objekte den Erkenntnisgegenstand.

> Es handelt sich um ein Combining, wenn folgende Satzzusammensetzung (englisch) vorliegt: *Subject can Activity Object*.

Dabei bildet „Activity Object“ das Combining, während „Subject“ und „Object“ konkrete Instanzen (Cells) darstellten.

3.3 Assoziationsklassen

Ontologien bieten die Möglichkeit Relationen zwischen den enthaltenen Terminologien anzulegen, deren Aufgabe die Beschreibung von Zusammenhängen darstellt. Diese Relationen können auf beliebige Weise definiert werden. Folgendes Beispiel dient zur Verdeutlichung:

> In einer Ontologie seien die beiden Klassen *Mensch* und *Fahrzeug* definiert. Weiterhin sind zwei Instanzen durch *Mensch(Bob)* und *Fahrzeug(LKW)* vorhanden. Eine Relation zwischen Bob und LKW sei definiert als *fahren(Bob, LKW)*.

Durch diese freie Definition von Relationen wird ein einheitliches „Durchlaufen“ auf Pfaden des Ontologiegraphen[3] und damit die Möglichkeit komplexe Schlussfolgerungen auf einfache Weise ziehen zu können, erschwert. Grund dafür ist die fehlende Formalisierung bei der Erstellung von Konzeptzusammenhängen. Um diesen Nachteil zu umgehen, verwendet das vorliegende Modell eine Menge vordefinierter Verbindungstypen, die als Assoziationsklassen bezeichnet werden. Im Gegensatz zur herkömmlichen Ontologie, bei der Zusammenhängen mit Tätigkeitscharakter über Kanten ausgedrückt werden, dienen hier Aktivitäten (Abschnitt 3.2.40., Seite 51), also Knoten, zur Repräsentation dieses Wissens. Auf diese Weise kann die Integration von Inferenzalgorithmen, beispielsweise zur Deduktion (Abschnitt 4.1, Seite 71), stark vereinfacht werden.

Die folgenden Abschnitte beschreiben die derzeitigen Assoziationsklassen genauer und beziehen sich dabei auf Beispiele ihrer Anwendung.

---

[3] mathematische Betrachtung des Graphen, der sich aus den Elementen und Relationen der Ontologie ergibt

3.3.1 Assoziationstypen: optionale und verpflichtende Kanten

Das Modell unterscheidet zwischen optionalen und verpflichtenden Assoziationen zwischen Elementen. Somit können optionale Elemente in die Deduktion einbezogen oder von ihr vernachlässigt werden, zum Beispiel durch Befragung der Nutzerbedürfnisse. Ein optionaler Pfad ist eine Kante zu einem Knoten, der während eines Schlussfolgerungsprozesses interessant sein kann, dessen tatsächliche Notwendigkeit aber erst zur Ausführungszeit des Deduktionsprozesses erkennbar ist. Die Menge aller optionalen Kanten wird definiert als $E_{opt}(G) = \{e_x, \ \ldots, \ e_n\}$.

Im Gegensatz zu optionalen Assoziationen, sind verpflichtende Kanten bindend für jeden beliebigen Schlussfolgerungsprozess, der die an den Kanten anliegenden Knoten durchlaufen muss. Dieser Typ entspricht dem Normalfall einer Kante und ist definiert in der Menge: $E_{req}(G) = \{e_x, \ \ldots, \ e_n\}$.

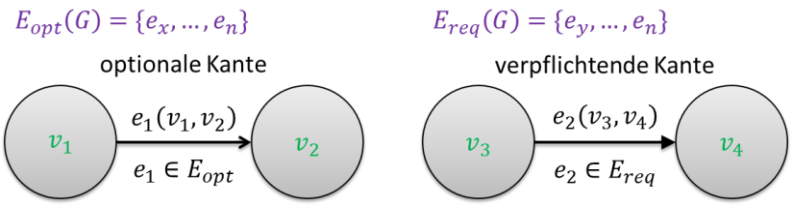


Abb. 17 Visualisierung von optionalen und verpflichtenden Kanten

Abb. 17 zeigt, wie die beiden Kantentypen visualisiert werden. Sie unterscheiden sich am Pfeilende: Verpflichtende Kanten sind durch eine ausgefüllte Pfeilspitze gekennzeichnet.

3.3.2 Basis-Assoziationsklassen: is, has, can, part-of und used-for

Neben der Typisierung von Assoziationen in verpflichtende und optionale Kanten, stehen im Modell verschiedene Assoziationsklassen zur Verfügung. Durch unterschiedliche Fallstudien haben sich fünf Basis-Assoziationsklassen als besonderes bedeutend herausgestellt (Abb. 18).

is: Vererbung und Instanziierung

Die is-Assoziation (Abb. 18, a) wird genutzt um Vererbungsstrukturen, Klassenzugehörigkeiten oder Instanziierungen im Modell zu integrieren. Zum Beispiel könnte die Beziehung zwischen „MS Visio“ und „Software“ mit einer is-Kante verdeutlicht werden: *MS Visio is*

Software. Diese Assoziationsklasse ist grundsätzlich an eine verpflichtende Kante gebunden. Der Grund: Die Entscheidung, ob ein bestimmtes Element ein Abbild oder ein Kind eines anderen ist, kann nicht optional vorliegen.

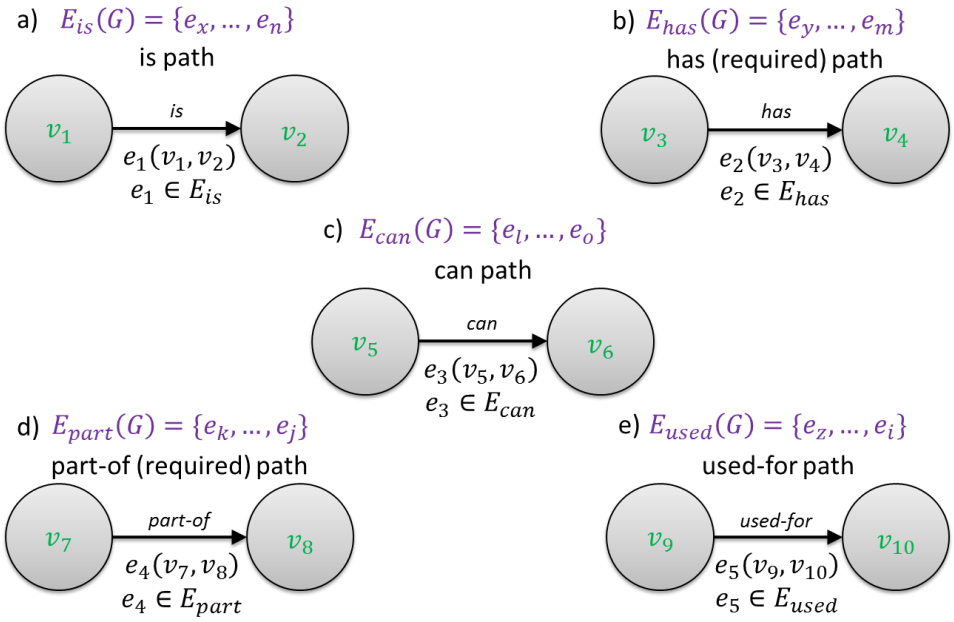


Abb. 18 Assoziationsklassen des Modells

used-for: Darstellung von Verwendbarkeiten

Auch Pfade, die eine used-for-Beziehung zwischen Elementen repräsentieren, sind verpflichtende Kanten. Diese Assoziationsklasse wird beispielsweise verwendet, um Activities von bestimmten Objekten oder Subjekten ausführbar zu machen. Sinngemäß kann die Beziehung zwischen den beiden Elementen aus Abb. 18, e) formuliert werden als: Element v_9 kann dazu genutzt werden, Element v_{10} zu aktivieren oder auszuführen, beispielsweise *MS Word used-for Textbearbeitung*.

has: Modellierung von Eigenschaften

Zur Modellierung von Eigenschaften dient die has-Assoziationsklasse. Dieser Beziehungstyp wird verwendet, um Elemente genauer zu spezifizieren und ihnen Attribute oder Merkmale zuzuweisen. Beispielsweise hat jede Software einen Preis und einen Installationstyp. Bei der Modellierung des Wissens sind also drei Elemente erkennbar: *Software*, *Preis* und *Installationstyp*. Zudem ergibt sich folgender Zusammenhang: *Software has Preis* und *Software has Installationstyp*. Infolgedessen bekommt jede Instanz von Software, also jeder „is“-Vorgänger, die Eigenschaften *Preis* und *Installationstyp* vererbt.

part-of: Teil-Ganzes-Beziehungen

Abb. 18, d) stellt eine weitere Assoziationsklasse dar, die part-of-Kante. Diese Klasse dient zur Modellierung von Teil-Ganzes-Beziehungen. Im Gegensatz zur has-Verknüpfung, bei der die verknüpften Elemente einzeln existieren können, verdeutlicht die part-of-Beziehung eine bindende Zusammengehörigkeit. Demzufolge kann v_7 nur bestehen, wenn auch v_8 existiert. Als Beispiel dient die Teil-Ganzes-Beziehung zwischen Display und Notebook: *Display part-of Notebook.*

can: Zuweisung von Fähigkeiten

Die letzte Basis-Assoziationsklasse bildet die can-Beziehung (Abb. 18, c). Sie zielt auf die Modellierung von Fähigkeiten eines Objekts oder Subjekts ab und stellt den häufigsten Verknüpfungstyp zwischen Cells und Activities bzw. Combinings dar. Durch diese Klasse können beispielsweise bestimmte Rollen mit Aufgaben versehen werden, Subjekte verschiedene Qualifikationen erhalten oder Objekten Funktionen und Befähigungen zugewiesen werden. Folgende Beispiele dienen zum Verständnis: *Mensch can denken. Licht can leuchten. Google can suchen. Jira can ‚Anforderungen organisieren'. ‚Product Owner' can ‚Anforderungen priorisieren'.*

3.3.3 Erweiterte Assoziationsklassen

Es ist modelltechnisch problemlos möglich, neben den Basis-Assoziationsklassen, beliebig viele Arten von Relationen für die Modellierung vorzugeben. Zur Integration komplexer Wissensstrukturen haben sich beispielsweise fünf weitere Verbindungsarten als notwendig herausgestellt. Diese können bei der Ontologieerstellung zur Wissensüberführung von Nutzen sein, werden jedoch im Rahmen dieser Arbeit für die Deduktionsalgorithmen aus Abschnitt 4.1 (ab Seite 71) nicht genauer betrachtet. Folgende Assoziationsklassen dienen zur erweiterten Modellierung von Wissen:

does: Beschreibung eines zeitlich konstanten Zustands

Diese Klasse wird benötigt, um Zustände zu einem bestimmten Zeitpunkt T_x in das Modell zu integrieren. Hintergrund ist die Wissensüberführung von statischen Gegebenheiten, die zu einer beliebigen Zeit zutreffend sind. Eine does-Relation erweitert demnach die can-

Assoziation um eine zeitliche Komponente, da die can-Assoziation Potenziale und keine zeitlichen Zustände im Modell hinterlegt.

Beispiel: Der Eifelturm steht in Paris.

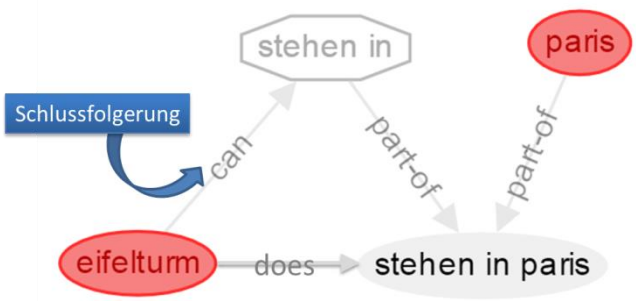


Abb. 19 does-Assoziation Beispiel: Eifelturm

Bei diesem Beispiel existieren vier Knoten im Modell: die Cell *eifelturm*, die Activity *stehen in*, die Cell *paris* und ein Combining aus Activity und Objekt *stehen in paris*. Abb. 19 zeigt, wie der Zusammenhang zwischen diesen Elementen modelliert wird. *stehen in* und *paris* sind bindende Teile des Combinings *stehen in paris*, weshalb sie in part-of-Beziehung zu diesem assoziiert wird. Dass der Eifelturm zum derzeitigen Zeitpunkt in Paris steht, beschreibt einen zeitlichen Zustand. Der Verbindungstyp muss demnach lauten: *eifelturm does stehen in paris*. Daraus ergibt sich die Schlussfolgerung, dass *eifelturm* die Fähigkeit besitzt, in etwas zu stehen (*stehen in*). Es kann also geschlussfolgert werden *eifelturm can stehen in*.

***-not: Negation von Aussagen**

Die Annahme, dass jeder Zusammenhang aus den in der Ontologie enthaltenen Wissenselementen möglich sei, solange nicht explizit etwas anderes modelliert wurde, wird als Open World Assumption (OWA) bezeichnet.[4] Wie in Abschnitt 2.3.3 (Seite 25) erwähnt, verwenden die meisten OWL-basierte Ontologien dieses Prinzip. Auch das vorliegende Modell zur Erstellung einer Ontologie verfolgt das Prinzip der OWA. Seien beispielsweise zwei Objekte *Kunststoff* und *leitfähig* in einer Ontologie enthalten. Dann lässt sich die Fra-

[4] Im Gegensatz zur OWA steht das Prinzipt der Closed World Assumption (CWA). Dabei wird Wissen nur aus den exakt formulierten Zusammenhängen abgeleitet.

ge „Ist Kunststoff leitfähig?“ nicht beantworten, solange keine konkrete Beziehung zwischen *Kunststoff* und *leitfähig* existiert. Um dieses Wissen aussagefähig zu machen, sind weitere Assoziationsklassen notwendig, die einen Zusammenhang zwischen Elementen negieren. Dazu dient die *-not-Relation. Das Asterix steht dabei für eine beliebige Assoziationsklasse, wie can-not, is-not oder not-part-of. Wird zwischen *Kunststoff* und *leitfähig* nun ein Zusammenhang modelliert *kunststoff has-not leitfähig*, wird die Eigenschaft *leitfähig* aus der Gesamtheit aller *Kunststoff*-Eigenschaften ausgeschlossen. Dann kann die eingangs gestellte Frage „Ist Kunststoff leitfähig?“ mit Nein beantwortet werden.

must: Zuweisung von Activities

Des Weiteren wird eine Verbindungsklasse benötigt, die Abhängigkeiten zwischen einer Activity und einem spezifischen Element erzwingt. Auf diese Weise können zum Beispiel bestimmten Rollen eines Software-Entwicklungsprozesses Aufgaben zugewiesen werden. Dieser Verbindungstyp entspricht der must-Assoziationsklasse. Während eine can-Verbindung eine gegebene Fähigkeit eines Objektes oder Subjektes beschreibt, wird durch must eine Fähigkeit erzwungen.

Beispiel: Product Owner und die mit dieser Rolle verknüpften Aufgabe, Anforderungen zu priorisieren.

Der Zusammenhang zwischen der Scrum-Rolle und der Aufgabe kann modelliert werden als *product owner must anforderungen priorisieren.* Soll beispielsweise das Projektmitglied *anna* die Rolle *product owner* verkörpern, muss folgender Zusammenhang gewährleistet werden, um dieser must-Verknüpfung gerecht zu werden: *anna can anforderungen priorisieren*. Existiert dieser Zusammenhang nicht und *anna* wird die Rolle dennoch zugewiesen, kann auf eine can-Verknüpfung zwischen *anna* und *anforderungen priorisieren* geschlussfolgert werden.

should: Integration „lockerer“ Zuweisung

Ähnlich zur must-Klasse verhält sich eine should-Assoziation. Durch diese Klasse können Zusammenhänge zwischen Elementen modelliert werden, die nicht zwingend zutreffen müssen, deren Vorhandensein aber eine Bereicherung darstellt. Dies wird beispielsweise benötigt, um moralische, ethische oder politische Zusammenhänge in die Ontologie zu integrieren. Als Beispiel dient das Programmierparadigma „Modulare Programmierung“:

programmierer should modular programmieren. Dabei stellt *modular programmieren* ein Combining aus der Cell *modular* und der Activity *programmieren* dar.

same-as: Synonyme und Sprachen

Insbesondere zur Integration von Synonymen, also zur Darstellung inhaltlich gleicher Information, die syntaktisch unterschiedlichen Terminologien entsprechen, wird eine weitere Assoziationsklasse notwendig, die same-as-Assoziation. Sie erlaubt die Definition von Elementen mit gleicher Bedeutung. Daher dient sie dem Modell auch zur Integration verschiedener Sprachen. Beispiele für diesen Verbindungstypen sind *festplatte same-as hard-disc-drive, programmieren same-as implementieren* oder *laptop same-as notebook.*

3.4 Erzeugen von Ergebnissen

Um Schlussfolgerungen in eine Ontologie zu integrieren, existieren verschiedene Ansätze. Die meisten basieren auf einem sogenannten Reasoner, der Axiome und andere Berechnungsgrundlagen nutzt, um Problemlösungen bereitzustellen. Abschnitt 2.3.5 (Seite 36) gibt einen Einblick in die Techniken des Ontology Learnings (Generierung von strukturiertem Wissen aus Textquellen) und in die Funktionsweise eines Reasoner. Auch in diesem Modell sollen Rückschlüsse auf Basis des im Modell enthaltenen Wissens durchgeführt werden können.

Daher besitzt jeder Knoten ein spezifisches Ergebnis, das bei seinen Nachbarzellen in die Berechnung ihres eigenen Ergebnisses einfließen kann. Das Ergebnis eines Knotens $v \in V$ wird definiert als $r(v)$. Die Berechnung eines Ergebnisses kann auf unterschiedliche Weise erfolgen. Beispielsweise könnte es durch eine lineare Funktion bestimmt werden, deren Variable die Abhängigkeit zu einem benachbarten Ergebnis darstellt:

$$r(v_x) = r(v_y) - 4 \quad mit\ v_y \in N_G(v_x)$$

Dabei bildet $N_G(v_x)$ die Menge aller Nachbarknoten von v_x. Darüber hinaus ist es möglich, ganze Knoten oder Knotenbezeichnungen in das Ergebnis eines bestimmten Elementes zu integrieren, zum Beispiel:

$$r(v_x) = \{v_a, v_b, v_c\}\ with\ v_a, v_b, v_c \in N_G(v_x)$$

In den folgenden Abschnitten wird die Erstellung des Ergebnisses in Abhängigkeit der verschiedenen Elementklasse beschrieben. Zudem erfolgt die Erläuterung von Bedingungsintegration durch positive und negative Constraints und dem damit verbundenen Element Pruning während eines Deduktionsprozesses.

3.4.1 Constraints und Deductive Reasoning Element Pruning

Neben den Assoziationsklassen existieren im Modell zwei besondere Arten der Abhängigkeitsformulierung, die als Negative- und Postive-Constraints (Bedingungen) definiert werden. Mit ihrer Integration ist das Ziel verbunden, den Ontologiegraphen effizienter zu durchlaufen, indem das Abarbeiten der Elemente intelligent erfolgt und somit Ergebnisse mit einer geringeren Anzahl an Deduktionsschritten abzuleiten.

Negative-Constraints

Negative-Constraints können als Schranken verstanden werden, die dazu dienen, einen Pfad innerhalb des Ontologiegraphen aus der Menge aller Pfade auszuschließen. Auf diese Weise können Elemente mit festgelegten Bedingungen bzw. Voraussetzungen mit anderen Elementen (z.B. Feature) verbunden werden. Eine Abarbeitung des Elementes ist demnach nur erforderlich, wenn all seine Voraussetzungen erfüllt sind.

Wenn mindestens ein Negative-Constraint eines Elementes *false* ist, kann das Element aus der Menge aller für eine mögliche Deduktion relevanten Knoten ausgeschlossen werden. Im Rahmen dieser Arbeit führt die Integration von Negative-Constraints beispielsweise schneller zu einer Liste mit möglichen Artefakten, die für die Umsetzung eines Softwareprojektes als besonders geeignet gelten. Wie Negative-Constraints im Modell modelliert werden zeigt Abb. 20. Die Menge aller Negative-Constraints wird definiert als $\Gamma(\mathrm{G}) = \{\gamma_1, \dots, \gamma_n\}$.

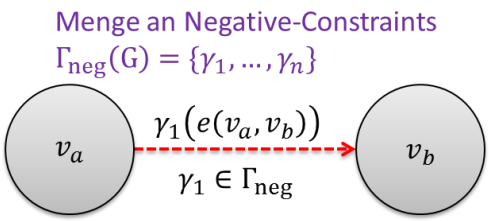
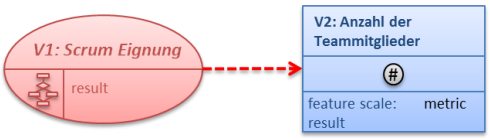


Abb. 20 Notation von Negative-Constraints

Ein einfaches Beispiel zur Anwendung von Negative-Constraints wird in Abb. 21 dargestellt. Darin enthalten ist eine Bedingung zwischen der Cell *V1: Scrum Eignung* und dem Feature *V2: Anzahl der Teammitglieder*. Eine Negative-Constraint γ wird der Kante $e(V1, V2)$ zugewiesen als $\gamma\big(e(V1, V2)\big)$: $r(V2) < 5 \vee r(V2) > 9$. Dies bedeutet, das Ergebnis von $V1$ wird false, also $r(V1) = false$, wenn das Ergebnis von $V2$ kleiner als fünf oder größer als neun ist.



$$\gamma\big(e(V1, V2)\big)\colon r(V2) < 5 \vee r(V2) > 9$$
$$mit\ r(V2) \in \mathbb{Z}\ und\ \gamma \in \Gamma_{\text{neg}}$$

Abb. 21 Beispiel einer Negative-Constraint

Ein sehr einfaches Praxisbeispiel lässt sich auf die Entscheidung zurückführen, welches Kleidungsstück getragen werden soll. In der Annahme, eine Software soll zwischen verschiedenen Jackenarten unterscheiden, sind mit dem Tragen dieser Jacken unterschiedliche Einflussfaktoren verknüpft. Die Betrachtung, ob eine Winterjacke sinnvoll ist, macht beispielsweise nur dann Sinn, wenn die Außentemperatur weniger als 4° C beträgt. Eine Negative-Constraint der Winterjacke in Richtung der Temperatur könnte also lauten:

$$\gamma\big(e(Winterjacke, Außentemperatur)\big)\colon r(Außentemperatur) \leq 4$$

Aus Sicht der Software würde es nun zu einer Ineffizienz führen, bei einer Außentemperatur von 10° C die Winterjacke als engere Auswahl zu prüfen.

Folgende Beispiele sollen verdeutlichen, dass Negative-Constraints nicht invertierbar sind:

- ***r(apfel) = false ← r(bananenförmig) == true***
  Das Ergebnis von *apfel* wird *false*, wenn das Ergebnis von bananenförmig *true* ist. Die Umkehrung dieses Arguments ist nicht möglich: Wenn *bananenförmig false* ist, ist *apfel* nicht automatisch *true*

- ***r(apfel) = false ← r(essbar) == false***
  Das Ergebnis von *apfel* ist *false*, wenn das Ergebnis von *essbar false* ist. Die Um-

kehrung dieses Arguments ist nicht möglich: Wenn etwas *essbar* ist, muss es sich nicht automatisch um einen *apfel* handeln.

Positive-Constraints

Die zweite Abhängigkeitsart bei der Ergebniserstellung bilden die Positive-Constraints. Der Unterschied beider Bedingungen liegt in der Verknüpfung der Ausdrücke. Bereits das Nicht-Erfüllen nur einer Negative-Bedingung führt zur Negation (*false*) eines Ergebnisses. Anders verhält es sich bei Positive-Constraints. Hierbei müssen alle Bedingungen wahr sein, damit das Ergebnis des sie enthaltenden Knotens *true* wird.

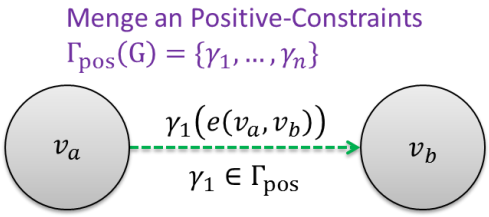


Abb. 22 Notation einer Positive-Constraint

Abb. 22 zeigt, wie Positive-Constraints im Modell verwendet werden und enthält die Mengen-Notation $\Gamma_{pos}(G)$, die alle Positive-Constraints beinhaltet.

Das Beispiel aus Abb. 23 veranschaulicht die Anwendung einer Positive-Constraint zwischen den Knoten *V1: Scrum Eignung* und *V2: Anzahl an Teammitgliedern*. Die Aussage hinter dieser Constraint lautet: Das Ergebnis von $V1$ ist *true*, wenn das Ergebnis von $V2$ größer als fünf und kleiner als neun ist.

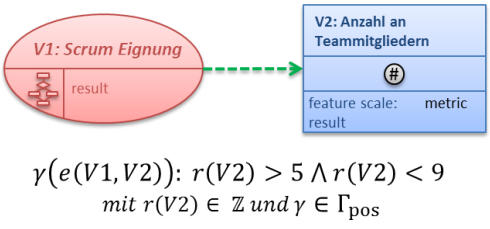


Abb. 23 Beispiel einer Positive-Constraint

Im Gegensatz zu Negative-Constraints, sind Positive-Constraints invertierbar. Dies wird anhand folgender Beispiele deutlich:

- ***r(apfel) = true ← r(rund) == true***
  Das Ergebnis von *apfel* ist *true*, wenn das Ergebnis von *rund true* ist. Die Umkehrung des Arguments ist möglich: wenn *rund false* ist, ist es kein *apfel*.

- ***r(apfel) = true ← r(essbar) == true***
  Das Ergebnis von *apfel* ist *true*, wenn das Ergebnis von *essbar true* ist. Wenn das Ergebnis von *essbar false* ist, ist auch *apfel false*.

Deductive Reasoning Element Pruning

Durch das Hinzufügen von Constraints lässt sich die Durchlaufzeit des Graphen verringern. Pfade, deren Bedingungen nicht erfüllt sind, können aus der Gesamtheit aller Pfade ausgeschlossen werden. Auf diese Weise wird der Deduktionsprozess effizienter, da weniger Elemente auf Eignung geprüft werden müssen. Weiterhin kann durch das Auslassen von Knoten während der Deduktion die Interaktion mit dem Nutzer verringert werden.

Dieser Vorgang der Knotenreduktion wird als „Deductive Reasoning Element Pruning (DREP)“ bezeichnet. Element Pruning ist ein aus der Literatur bekannter Begriff beispielsweise für Selektionsalgorithmen (DOR UND ZWICK 1999; DOR 1995; ANAND UND GUPTA 1998) oder aus Clustering Methoden wie in (BISSON ET AL. 2000) oder (FERNANDES UND GARCÍA 2012). DREP ist demnach eine Optimierungsmaßnahme während der Deduktion, die auf Basis von Constraints zu einem bestimmten Zeitpunkt T die Anzahl der zu durchlaufenden Restelemente des Zeitpunkts $T + 1$ verringern kann.

Durch die Integration von DREP kann Parallelität zu Knotensprüngen führen, deren Pfade unter Umständen gar nicht mehr erreichbar sind, da sie durch Pruning abgetrennt wurden. Dies geschieht durch Pruning von Brückenkanten. Brücken, im Sinne der Graphentheorie, sind Kanten bei deren Trennung mindestens ein Weg des Graphen nicht mehr möglich ist. Unter Einbezug von Pruning beschleunigt parallele Abarbeitung demnach lediglich den *vollständigen* Deduktionsprozess, d.h. nur bei vollständigem Durchlaufen des Graphen ist paralleles Vorgehen sinnvoll. Für den Fall der Pruningintegration sollte daher entweder vollständig auf Parallelität verzichtet oder Gebrauch von einer eventbasierten Abarbeitung gemacht werden.

3.4.2 Ergebniserzeugung der einzelnen Elementklassen

Die Erzeugung der Ergebnisse ist abhängig von der Klasse des Elementes, dessen Ergebnis erzeugt werden soll. Zusätzliche Bedingungen, die durch Negative- oder Positive-Constraints definiert sind, beeinflussen ebenfalls die Erzeugung des Ergebnisses. Folgende Aussagen stellen die Kern-Methodiken zur Generierung der Ergebnisse dar:

Die Ergebnisse von Items und Cells sind Boolean (*true*, f*alse*) bzw. (*0, 1*) oder leer:

$$r(v_x) \in \{0,1,\emptyset\}\ mit\ v_x \in L_1 \cup L_2$$

Items sind *true*, wenn mindestens einer der adjazenten is-Vorgänger *true* ist:

$$r(v_x) = 1 \Leftrightarrow \exists v_y = 1\ mit\ v_y \in N_{is}^{-}(v_x)\ und\ v_x \in L_1$$

Cells besitzen boolesche Ausdrücke um ihre Ergebnisse zu erstellen. Diese Ausdrücke hängen von den Negative- und Positive-Constraints einer Cell ab:

$$r(v_x) = \Gamma_{pos}(v_x) \bigwedge ! \Gamma_{neg}(v_x)\ mit\ v_x \in L_2$$

Demnach sind Cells *true*, wenn das Ergebnis jedes Negative-Constraint-Nachfolgers *false* ist und das Ergebnis jedes Positive-Constraint-Nachfolgers *true* ist:

$$\forall v_x \in L_2 : r(v_x) = 1 \Leftrightarrow \forall \gamma_y \in \Gamma_{pos}(v_x) : \gamma_y = 1\ und\ \forall \gamma_z \in \Gamma_{neg}(v_x) : \gamma_z = 0$$

Des Weiteren kann das Ergebnis eines Input-Features jeder mögliche Eingabewert sein. Ein Choice-Feature kann das Ergebnis jeder adjazenten Cell setzen, solange sie part-of-Vorgänger dieses Features ist. Questions, als besondere Form eines Data-Source-Elementes, haben kein Ergebnis. Sie dienen lediglich der Ausgabe einer Fragestellung, um Nutzereingaben durch Feature genauer abfragen zu können.

3.5 Anwendungsbeispiele

Um die Anwendung des vorgestellten Modells zu verdeutlichen, erfolgt im Folgenden die Darstellung und Beschreibung ausgewählter Beispiele. Diese Beispiele orientieren sich nicht ausschließlich an Anwendungsfällen des Software Engineering. Um die Verwendung der einzelnen Modellkomponenten schrittweise zu demonstrieren, werden zum Teil sehr

einfache Wissenszusammenhänge modelliert. Entsprechend nimmt die Komplexität des enthaltenen Wissens mit jedem Beispiel zu, bis schließlich ein anwendungsnahes Beispiel aus der Softwareentwicklung die Praxistauglichkeit des Modells verdeutlichen soll.

Beispiel I: Elemente- und Assoziationsklassen, Abstraktion und Konkretisierung

Abb. 24 zeigt, wie die verschiedenen Element- und Assoziationsklassen anhand eines einfachen Beispiels grafisch abgebildet werden. Die Grafik verdeutlicht einfache Wissenszusammenhänge zwischen Frucht, Apfel und deren Eigenschaften. Die abstrakte Ebene (Items) wird durch die gelben Rechtecke symbolisiert. Dabei stellt *frucht* ein Konzept dar, welches zwei Eigenschaften besitzt: *farbe* und *form*. Konkrete Instanzen (Cells), dargestellt durch rote Ellipsen, einer *farbe* sind *blau*, *rot* und *grün*. Von *frucht* existiert im Beispiel nur *apfel* als Konkretisierung. Neben konkreten Farben existieren die konkreten Formen *rund* und *eckig*.

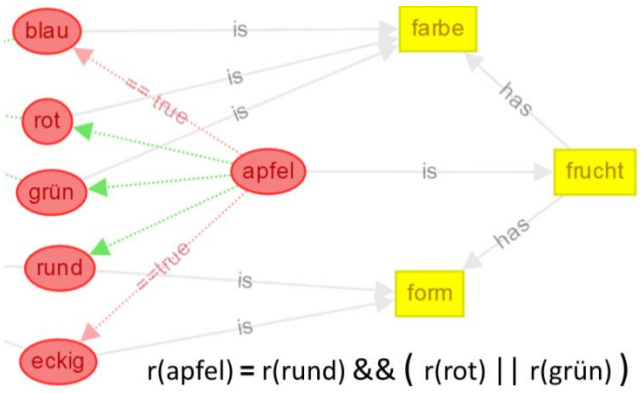


Abb. 24 Elemente- und Assoziationsklassen, Abstraktion und Konkretisierung

Da der hier modellierte Apfel die Instanz einer Frucht darstellt, bekommt er ihre Eigenschaften vererbt. Als Instanz besitzt er, im Gegensatz zur Frucht, jedoch keine abstrakten Eigenschaften, sondern deren konkrete Ausprägungen. Er muss also in Zusammenhang mit den Farben rot, blau oder grün und den Formen rund oder eckig gebracht werden. Diese Abhängigkeiten werden im Beispiel durch Positive- und Negative-Constraints ausgedrückt. *apfel* besitzt zwei Negative-Constraints. Eine zur Farbe *blau* und eine zur Form *eckig*. Sollte eines der beiden Elemente zutreffen, also sein Ergebnis „true“ entsprechen, führt dies zur Negation von *apfel*. Vereinfacht ausgedrückt: Ist das zu untersuchende Objekt *blau* oder *eckig*, handelt es sich nicht um einen *apfel*. Positive-Constraints besitzt *apfel* zu den modellierten Farben *rot* und *grün*, sowie zur Form *rund*. Diese ergeben sich aus dem boo-

leschen Ausdruck, welcher dem Ergebnis von *apfel* zugewiesen ist. Er besagt, *apfel* wird true, wenn *rund* und zusätzlich entweder *rot* oder *grün* zutreffen.

Beispiel II: Datenerhebung und Informationsverarbeitung

Das Beispiel aus Abb. 25 umfasst Entscheidungswissen zur Auswahl einer Jacke in Abhängigkeit von Temperatur und Wetter.

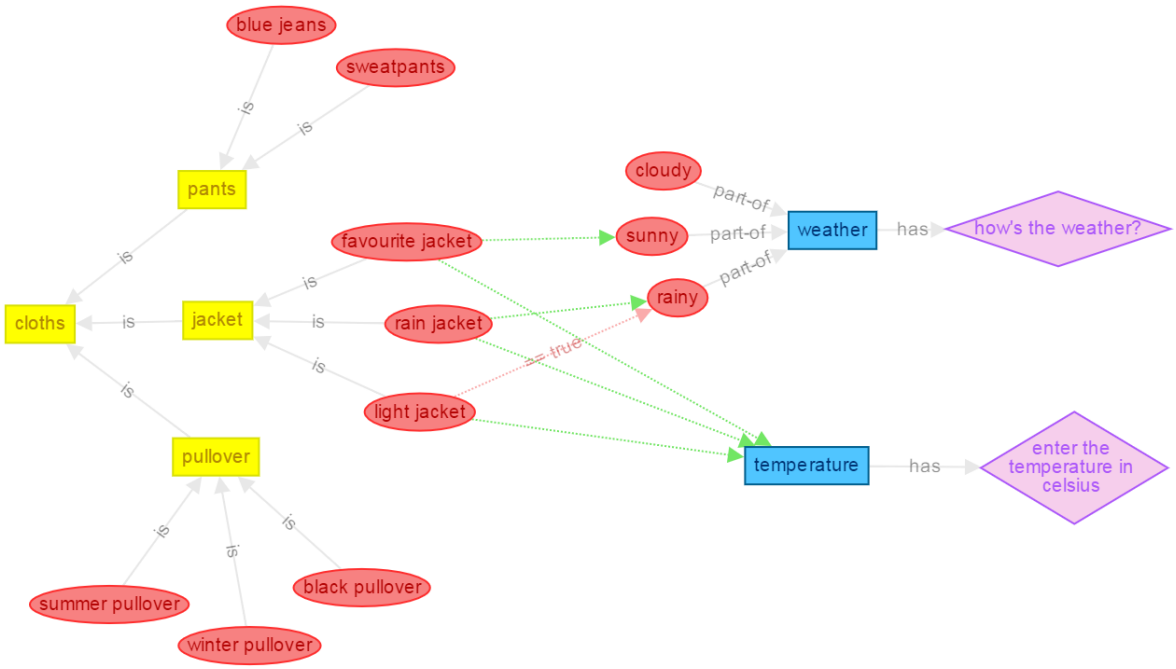


Abb. 25 Datenerhebung und Informationsverarbeitung

Wird zunächst die Modellierung von abstraktem und konkretem Wissen in diesem Beispiel betrachtet, können folgende Aussagen getroffen werden:

- Es existiert ein abstraktes Konzept *cloths*, von der es drei abstrakte Ausprägungen gibt: *pants*, *jacket* und *pullover*.

- Es sind 10 Cells modelliert, von denen *favourite jacket*, *rain jacket* und *light jacket* als konkrete Instanzen von *jacket* existieren.

Um Aussagen treffen zu können, die in Abhängigkeit zu dynamischen Einflussgrößen, wie Wetter oder Temperatur stehen, bietet das Modell Feature und Data Source-Elemente (Abschnitt 3.2.2, S. 48). Diese Elemente bilden die (Daten-)Eingabeschicht des Modells und werden im Beispiel durch ein Input-Feature *temperature* und ein Single-Choice-Feature *weather* verkörpert. Auf eine visuelle Unterscheidung von Input-, Range-, Single-Choice-

und Multiple-Choice-Feature wurde verzichtet. *cloudy*, *sunny* und *rainy* sind Auswahlmöglichkeiten des Single-Choice-Feature *weather*. Um diese Cells zu aktivieren oder zu deaktivieren, erfolgt die mit dem Feature verbundene Ausgabe der Frage „how's the weather?". Der Nutzer wählt aus einer der drei Antwortmöglichkeiten die Passende aus. Ihr Ergebnis wird *true* gesetzt, die restlichen Ergebnisse *false.*

Wie eingangs erwähnt, soll bei der Jackenauswahl neben dem Wetter eine Abhängigkeit zur Temperatur erfolgen. Temperaturdaten können per Sensor gemessen oder wie hier modelliert per Eingabe vom Nutzer erfasst werden. Diese Daten fließen in die Ergebnisgenerierung der Jackeninstanzen ein. Das Ergebnis von *rain jacket* lautet:

$$r(\text{rain jacket}) = r(\text{rainy}) \;\&\&\; r(\text{temperature}) < 20$$

Damit *rain jacket* zutrifft, also das Ergebnis *true* wird, muss der Nutzer bei der Frage nach dem Wetter *rainy* ausgewählt haben und eine Temperatur unter 20°C eingegeben haben.

Die Entscheidung, welche Jacke gewählt werden soll fällt nach Eingabe aller zu erfassenden Daten und den Ausführungen aller Ergebnisse auf diejenige, deren Ergebnis *true* ist. Konfliktfälle werden in Abschnitt 4.2.2 (Seite 81) näher betrachtet.

Beispiel III: Modellierung komplexer Wissenszusammenhänge

Abb. 26 zeigt einen Auszug des in Abb. 47 (Anhang, Seite xx) modellierten komplexen Wissens über die Eignung von Scrum und verschiedenen damit assoziierbaren Begriffen. Mit diesem Wissen kann Scrum hinsichtlich verschiedener Einflussgrößen, wie Teamdisziplin oder Motivation, auf seine Eignung als Vorgehensmodell für ein Softwareentwicklungsprojekt geprüft werden. Zur besseren Orientierung wurde die Grafik in neun Felder untergliedert. Die Cell *scrum eignung* befindet sich im Feld 2C. Sie weist zahlreiche Abhängigkeiten zu unterschiedlichen Elementen und Elementklassen auf. *scrum eignung* selbst bildet eine Instanz der Klasse *eignung*, welche wiederum als Eigenschaft einer *software entwicklungsmethode* modelliert wurde. Als Instanz einer *software entwicklungsmethode* bekommt *scrum* (Grenze von 1C zu 2C) die konkrete *scrum eignung* als Eigenschaft vererbt. Das Ergebnis *r(scrum) = r(scrum eignung)* lässt darauf schließen, dass es sich immer dann um *scrum* handelt, wenn *scrum eignung* zutrifft.

Abhängigkeiten von *scrum eignung* ergeben sich insbesondere durch unterschiedliche Negative-Constraints. Bereits erläutert (u.a. in Abschnitt 3.4.1, Seite 60) führen diese zur Negation einer Cell. In diesem Fall beispielsweise bei *homogener* Wissenszusammensetzung (*interdisziplinäres Wissen*, Feld 3B) oder wenn von vornherein feststeht, dass Scrum als Entwicklungsprozess für das geplante Projekt nicht in Frage kommt (*vorhaben scrumnutzung*, Feld 3C und entsprechende Auswahlmöglichkeit „*nein*“, Feld 2C).

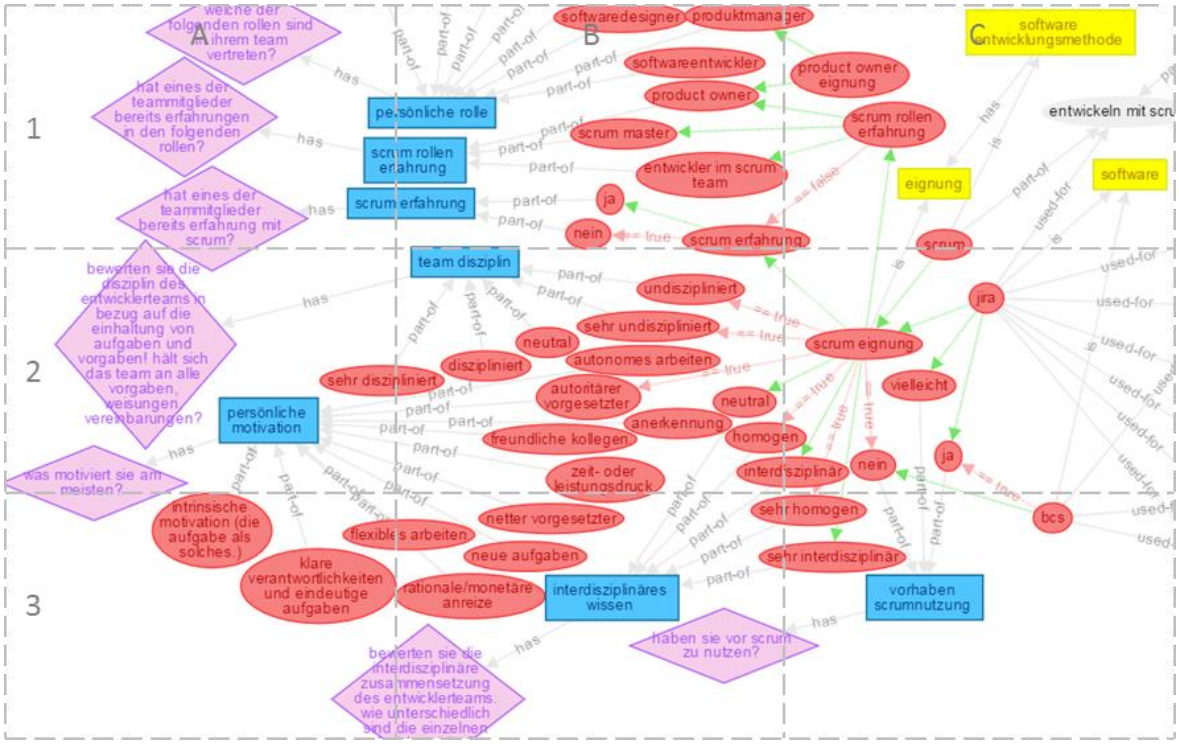


Abb. 26 Komplexes Wissen am konkreten Beispiel (Auszug)

3.6 Zusammenfassung: Abbildung von Wissen, Information und Daten

Das Modell bietet vier unterschiedliche Elementklassen zur Repräsentation von Informationen und Wissen. Combinings und Activities bilden erweiterte Elementklassen, die zur Integration von Fähigkeiten und zur Zusammensetzungen einzelner Wissenselemente dienen. Tabelle 1 enthält alle derzeit im Modell enthaltenen Elementklassen.

Tabelle 1 Elementklassen

	*Data Source*	Schnittstellen- bzw. Eingabeschicht zur Erfassung von Nutzer- und Sensordaten
	*Feature*	Überführung von Daten zu Informationen und Wissen
	*Cell*	A-Box: Integration von konkreten Instanzen, Objekten, Subjekten
	*Item*	T-Box: Integration von abstrakten Begriffen, Klassen, Konzepten
	*Combining*	Zusammenschluss von Elementen
	*Activity*	Integration von Fähigkeiten, Tätigkeiten und Aufgaben

Weiterhin stehen dem Modellierer unterschiedliche Assoziationsklassen zur Verfügung, mit denen Zusammenhänge zwischen den Wissenselementen abgebildet werden können. Die derzeit wichtigsten Assoziationsklassen sind in Tabelle 2 zusammengefasst.

Tabelle 2 Assoziationsklassen

Basis-Assoziationsklassen	
is	Vererbung und Instanziierung
has	Modellierung von Eigenschaften
can	Zuweisung von Fähigkeiten
part-of	Teil-Ganzes-Beziehungen
used-for	Darstellung von Verwendbarkeiten
Erweiterte Assoziationsklassen	
**-not*	Negation von Aussagen
should	Integration „lockerer“ Zuweisung
does	Beschreibung eines zeitlich konstanten Zustands
must	Zuweisung von Activities
same-as	Synonyme und Sprachen

Die Notation von Assoziations- und Elementklassen orientiert sich anhand bekannter Schreibweisen aus Mengen- und Graphentheorie. Einen Auszug aus den verwendeten Schreibweisen bietet Tabelle 3, welche die Notationen wesentlicher adjazenter Knoten enthält.

Ziel ist es, die Erstellung von Ontologien anhand eines abstrakten Modells zu erleichtern, indem komplexe Sachverhalte aus dem Bereich des Software-Engineering auf einfache Weise abgebildet werden können. Die Komponenten des Modells dienen zur Modellierung von Daten und deren Erfassung bis zur Darstellung von Informationen und Wissen (Abb. 46, Anhang S. xix).

Tabelle 3 Notationen adjazenter Knoten

$N_{is}^{-}(v_x)$	is-Vorgängerknoten des Knotens v_x
$N_{is}^{+}(v_x)$	is-Nachfolgerknoten des Knotens v_x
$N_{has}^{-}(v_x)$	has−Vorgängerknoten des Knotens v_x
$N_{has}^{+}(v_x)$	has-Nachfolgerknoten des Knotens v_x
$N_{part}^{-}(v_x)$	part-of-Vorgängerknoten des Knotens v_x
$N_{part}^{+}(v_x)$	part-of-Nachfolgerknoten des Knotens v_x
$N_{used}^{-}(v_x)$	used-for-Vorgängerknoten des Knotens v_x
$N_{used}^{+}(v_x)$	used-for-Nachfolgerknoten des Knotens v_x
$N_{can}^{-}(v_x)$	can-Vorgängerknoten des Knotens v_x
$N_{can}^{+}(v_x)$	can-Nachfolgerknoten des Knotens v_x
$N_{pos}^{-}(v_x)$	positive constraint-Vorgängerknoten des Knotens v_x
$N_{pos}^{+}(v_x)$	positive constraint-Nachfolgerknoten des Knotens v_x
$N_{neg}^{-}(v_x)$	negative constraint-Vorgängerknoten des Knotens v_x
$N_{neg}^{+}(v_x)$	negative constraint-Nachfolgerknoten des Knotens v_x

4 Inferenzansätze auf Basis des Modells

4.1 Schlussfolgerungen aus Deduktion, Assoziation und Vererbungsstruktur

Die Wissensbasis besteht aus Elementen, die zur Ableitung von Wissen sinnvoll miteinander kombiniert werden müssen. Es existieren Informationsfragmente unterschiedlicher Abstraktionstiefen, Daten und Assoziationstypen. Zudem ergeben sich aus der Basisarchitektur zwei Sichten auf das Wissenssystem: eine beschreibende Sicht, die unabhängig vom Betrachter Wissen erzeugen kann und eine entscheidende Sicht, die Wissen auf Grundlage von äußeren Einflüssen erzeugt. Es kann zudem eine Vermengung beider Sichten erfolgen, um beschreibende Elemente unter Einfluss unterschiedlicher Parameter eindeutig zu identifizieren. Um Wissen unter Einbezug von Sichten und Elementen ableiten zu können, sind unterschiedliche algorithmusbasierte Deduktionen notwendig. Basis dieser Deduktionsalgorithmen bilden die verschiedenen Assoziations- und Elementklassen des Models. Ziel ist, Schlussfolgerungen aus gegebenen Sachverhalten ableiten zu können, die sich, in Abhängigkeit variabler Einflussfaktoren, speziell auf eine gesuchte Menge von Elementen beziehen.

Aus den Zielen des Modells (Abschnitt 3.1, Seite 43) und den darin enthaltenen Fragestellungen lassen sich die in Tabelle 4 aufgelisteten verallgemeinerten Problemstellungen ableiten.

Tabelle 4 Problemstellungen der Deduktionsalgorithmen

1. Handelt es sich um ein Element einer gegebenen Sorte?
2. Um welches Element handelt es sich konkret?
3. Woraus besteht ein Element?
4. Welche Eigenschaften besitzt ein Element?
5. Finde Elemente einer bestimmten Klasse zur Erfüllung eines gegebenen Zwecks!

Diese Liste entspricht lediglich einem Auszug aller möglichen Fragestellungen, um model-

liertes Wissen abzuleiten. Es sind jedoch essenzielle Fragen, die dazu dienen Grundinformationen eines speziellen Sachverhaltes zu ermitteln.

Im Modell lassen sich zwei Klassen von Deduktionsalgorithmen unterscheiden: Konkretisierungs- und Abstraktionsalgorithmen. Beide Klassen beziehen Taxonomien und Vererbungsstrukturen, also hierarchische Zusammenhänge konkreter Instanzen und Konzepte in ihre Lösungssuche ein. Sie basieren beide auf der Untersuchung von Element- und Assoziationsklassen. Ihr Unterschied lässt sich auf eine bindende Existenz konkreter Elemente zurückführen: Während Konkretisierungsalgorithmen ihre Entscheidungen ausschließlich anhand mindestens eines konkret verfügbaren Elementes treffen können, sind Abstraktionsalgorithmen in der Lage, Fragestellungen auf Basis vollständig abstrakter Objekte oder Subjekte zu beantworten.

Zu Beginn jedes Deduktionsprozesses existiert eine nicht leere Menge $V(G)$, welche alle Knoten des Graphen G enthält. Weiterhin wird zu Beginn die endliche leere Knotenmenge M und die endlich leere Menge S benötigt.

$$\exists V(G) \neq \emptyset$$
$$\exists M(V) = \emptyset$$
$$\exists S(V) = \emptyset$$
$$M \subseteq V$$
$$S \subseteq V$$

Durch Integration von M kann jeder beliebige Knoten als Start- und Zielknoten bestimmt werden. Sie ist eine Teilmenge von V und dient während der Deduktionsprozesse zur sukzessiven Speicherung von Elementen. M ist demnach diejenige Menge, die alle zu durchlaufenden Elemente enthält. Durchlaufene Elemente sind Knoten, die sich auf dem Pfad während der deduktiven Inferenz befinden. Jedes Element aus V darf höchstens einmal in M vorkommen.

Bei den meisten hier aufgezeigten Algorithmen ist eine Lösungsmenge erforderlich, die alle gesuchten Elemente enthält. Diese wird als S definiert.

4.1.1 Konkretisierungsalgorithmen

Isn't-it?

Während die Fragen 2-5 aus Tabelle 4 mindestens ein Element in ihre Antwort einbeziehen müssen, führt die Beantwortung von Frage 1 jeweils zu einem simplen Ja oder Nein, was den Deduktionsalgorithmus als einfach erscheinen lässt. Der Algorithmus, zur Klärung dieser Fragestellungen wird als *Isn't-it?*-Deduktionsprozess bezeichnet und entspricht der Lösung eines einfachen Entscheidungsproblems. Konkrete Beispiele für „Handelt es sich um ein Element?" lauten: Handelt es sich um Scrum? Handelt es sich um ein erfolgreiches Projekt? oder Ist Frau Mustermann eine Java-Expertin?

Im Untersuchungsgegenstand dieser Arbeit kann der *Isn't-It?*-Deduktionsprozess als simpler Entscheidungsvorgang beschrieben werden, durch den das Ergebnis des gesuchten Elementes erzeugt und anschließend auf *true* oder *false* überprüft wird. Ziel dieses Algorithmus ist demnach die Ergebniserzeugung des gesuchten Elementes unter Berücksichtigung aller auf dem Ergebnispfad assoziierter Einflüsse (Feature).

Tabelle 5 enthält die neun Schritte, die für die Lösungssuche nach dem Isn't-It?-Algorithmus notwendig sind. Dabei entspricht v_0 nicht dem $v_0 \in L_0$ von Seite 46.

Tabelle 5 Isn't-It? Algorithmus

1	Speichere gesuchtes Element als $\boldsymbol{v_0}$ und setze den Zeiger $\boldsymbol{v_x = v_0}$
2	Füge v_x zu *M* hinzu: $M \cup \{v_x\}$ Wenn es sich bei v_x um ein Item handelt: $v_x \in L_1 \rightarrow Schritt\ 3$ Wenn es sich bei v_x um eine Cell handelt: $v_x \in L_2 \rightarrow Schritt\ 4$ Wenn es sich bei v_x um ein Feature handelt: $v_x \in L_3 \rightarrow Schritt\ 5$ Wenn es sich bei v_x um eine Datenquelle handelt: $v_x \in L_4 \rightarrow Schritt\ 6$
3	Führe für jeden is-Vorgängerknoten von v_x Schritt 2 aus: $\forall v_i \in N_{is}^{-}(v_x) \mid v_x = v_i$ $v_x \neq \emptyset \rightarrow Schritt\ 2$

4	Führe für jeden part-of-Nachfolgerknoten, wenn es sich dabei um ein Feature handelt, und für jeden Constraint-Nachfolgerknoten Schritt 2 aus: $\forall v_i \in N^+_{pos}(v_x) \cup N^+_{neg}(v_x) \cup \left(N^+_{part}(v_x) \cap L_3\right) \mid v_x = v_i$ $v_x \neq \emptyset \rightarrow Schritt\ 2$
5	Führe für jeden has-Nachfolgerknoten, wenn es sich dabei um eine Data Source handelt, Schritt 2 aus: $\forall v_i \in N^+_{has}(v_x) \cap L_4 \mid v_x = v_i$ $v_x \neq \emptyset \rightarrow Schritt\ 2$
6	Suche von allen Elementen ohne Ergebnis aus $M \cap L_3$ nach demjenigen mit der höchsten Anzahl an Eingängen (maximaler Eingangsgrad) und setze den Zeiger v_x auf dieses Element: $v_x \mid d^-_G(v_x) = \max\{d^-_G(v_i) \mid i \in \{1, \dots, n\};\ n = \lvert M \cap L_3 \rvert\};\ v_i, v_x \in M \cap L_3$ $v_x \neq \emptyset \rightarrow Schritt\ 7$
7	Versuche das Ergebnis von v_x zu erzeugen. Wenn Ergebnis vorhanden, weiter mit Schritt 8. Sonst fahre mit Schritt 9 fort: $r(v_x) \neq \emptyset \rightarrow Schritt\ 8$ $r(v_x) = \emptyset \rightarrow Schritt\ 9$
8	Wenn $v_x = v_0$ beende den Deduktionsprozess, sonst setze für jedes Constraint-Vorgängerelement von v_x, das sich in M befindet und für jedes is-Nachfolgerelement von v_x, das sich in M befindet den Zeiger v_x auf dieses Element und führe Schritt 7 aus: $v_x = v_0 \rightarrow Ende$ $\forall v_i \in \left(N^-_{neg}(v_x) \cup N^-_{pos}(v_x) \cup N^+_{is}(v_x)\right) \cap M \mid v_x = v_i$ $v_x \neq \emptyset \rightarrow Schritt\ 7$
9	Suche von allen Elementen, die sich in M befinden, noch kein Ergebnis haben und Constraint-Nachfolger oder is-Vorgänger von v_x sind nach demjenigen mit dem höchsten Grad, setze auf dieses Element den Zeiger v_x und führe Schritt 7 aus $A = M \cap \left(N^+_{neg}(v_x) \cup N^+_{pos}(v_x) \cup N^-_{is}(v_x)\right)$ $v_x \mid d^-_G(v_x) = max\{d^-_G(v_i) \mid i \in \{1, \dots, n\};\ n = \lvert A \rvert\}\ mit\ v_i, v_x \in A\ und\ r(v_i) = \emptyset$ $v_x \neq \emptyset \rightarrow Schritt\ 7$

Eine Unterscheidung auf Basis der Modellebenen ist erforderlich, da die einzelnen Elementklassen ihre Ergebnisse unterschiedlich bilden können. Beispielsweise können Cells bei der Erzeugung ihrer Ergebnisse boolesche Ausdrücke einbeziehen, die aus den Positive- und Negative-Constraints hervorgehen. Items hingegen sind beschreibende Elemente konkreter Gegebenheiten. Demzufolge handelt es sich genau dann um ein bestimmtes Item, wenn mindestens eine dem Item entsprechende Konkretisierung existiert. Elemente deren Negative-Constraints aufgrund eines adjazenten Ergebnisses nicht erfüllt sind, können ausgeschlossen werden (Deductive Reasoning Element Pruning, Abschnitt 3.4.1, Seite 60).

Kind-of?

Während der *Isn't-It?*-Algorithmus ein Element auf seine pure Existenz überprüft, kann der *Kind-Of?*-Algorithmus darüber hinaus Instanzen bzw. Ausprägungen eines Elementes finden. Damit soll Problemstellung 2 (Tabelle 4, S. 71) beantwortet werden: Um welches Element handelt es sich konkret? Dafür wird im ersten Iterationsschritt der zuvor beschriebene *Isn't-It?*-Algorithmus ausgeführt, um anschließend alle Kindelemente auszugeben, die einer is-Assoziation entsprechen und als Ergebnis *true* aufweisen. Der Ablauf ist in Tabelle 6 zusammengefasst.

Tabelle 6 Kind-of? Algorithmus

1	Führe „Isn't-It?"-Algorithmus für $\boldsymbol{v_x = v_0}$ aus. Das Einstiegselement entspricht dabei $\boldsymbol{v_0}$.
2	Wenn das Ergebnis von v_x true entspricht, besteht die Ergebnismenge aus allen is-Vorgängerelementen von v_x, deren Ergebnis ebenfalls true ist. Sonst kann der Deduktionsprozess beendet werden, da es sich nicht um das gesuchte Element handelt: $r(v_x) = 1 \rightarrow S \subseteq N_{is}^{-}(v_x) \vert v_i \in N_{is}^{-}(v_x)\ mit\ r(v_i) = 1$ $r(v_x) = 0 \rightarrow Ende$

Tabelle 21 Ergebnisse der Literaturrecherche verdeutlicht das Funktionsprinzip des Algorithmus. Beispielsweise könnte der Wurzelknoten *cloths* als Einstiegselement v_0 gewählt werden.

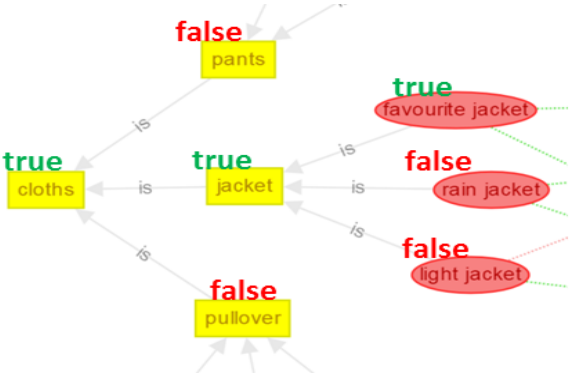


Abb. 27 Verständnisbeispiel zum Kind-Of?-Algorithmus

Nachdem der Algorithmus entsprechend der beschriebenen Schritte ausgeführt wurde, sollen die Ergebnisse der Kind- und Kindeskindknoten den in der Grafik ergänzten *true*- und *false*-Angaben entsprechen. Die Lösungsmenge *S* enthält dann das Element *jacket*. Würde der Einstieg bei *jacket* erfolgen, besteht die Lösungmenge *S* aus der Cell *favourite jacket*.

4.1.2 Abstraktionsalgorithmen

Parts?

Problemstellung 3 (Tabelle 4, S. 71) „Woraus besteht ein Element?“ bildet die Zielstellung dieses Abstraktionsalgorithmus. Er dient in erster Linie der Suche nach Teil-Ganzes-Beziehungen, um Fragestellungen, wie „Woraus besteht Software Engineering?“, beantwortet zu können. Wie Tabelle 7 zeigt, ist zur Lösung des Problems genau ein Schritt erforderlich. Grund hierfür ist das Modell selbst, welches bereits eine eigene Assoziationsklasse „part-of“ für Teil-Ganzes-Relationen bereitstellt.

Tabelle 7 Parts? -Algorithmus

1	Setze den Zeiger $\boldsymbol{v_x}$auf das gesuchte Element. $\boldsymbol{S}$ ist die Menge aller part-of-Vorgängerelemente von $\boldsymbol{v_x}$. $$\boldsymbol{S = N^-_{part}(v_x)}$$

Als Erweiterung dieses Algorithmus kann die Lösungsmenge anhand von Elementklassen gefiltert werden. Auf diese Weise können beispielsweise gezielt Aufgaben oder Fähigkeiten eines Objektes gefunden werden: „Welche Aufgaben sind mit Software Engineering verbunden?“ oder „Welche Aufgaben müssen für ein vollständiges Software Engineering erledigt werden?“. Die zur Filterung nach Activities und Combinings notwendigen Schritte sind in Tabelle 8 aufgelistet.

Tabelle 8 Parts?-Algorithmus Erweiterung

1	Setze den Zeiger $\boldsymbol{v_x}$ auf das gesuchte Element.
2	Setze für jeden part-of-Vorgänger von v_x den Zeiger v_x auf dieses Element und führe Schritt 3 aus. $\forall v_i \in N^-_{part}(v_x) \mid v_x = v_i$
3	Wenn v_x eine Aktivität oder ein Combining mit einer Aktivität ist, speichere v_x in S, sonst führe Schritt 2 aus. $v_x \in A \cup C \rightarrow S \cup v_x$ $v_x \notin A \cup C \rightarrow Schritt\ 2$

Characteristics?

Dieser Algorithmus findet Eigenschaften von Elementen und dient daher zur Lösung von Problemstellung 4 aus Tabelle 4, Seite 71. Seine Ausführung beantwortet Fragestellungen, wie „Welche Eigenschaften hat ein guter Entwickler?“, „Was zeichnet Konrad Zuse aus?“ oder „Was ist Scrum?“. Die Besonderheit hierbei bildet die notwendige Differenzierung zwischen Cells und Items. Eigenschaften von Cells entsprechen konkreten Instanzen, die per Positive- und Negative-Constraint-Pfaden ermittelt werden können. Items können zusätzlich Eigenschaften durch has-Assoziationen erhalten. Der Algorithmus sucht auch nach Klassenzugehörigkeiten, indem in Schritt 4 alle is-Nachfolger in die Ergebnisliste aufgenommen werden. Die unterschiedlichen Eigenschaften werden in Teilmengen von S gespeichert:

- S_{pos} ist die Menge, die alle Elemente enthält, die auf v_x zutreffen müssen.
- S_{neg} ist die Menge, die alle Elemente enthält, die auf v_x nicht zutreffen dürfen.
- S_{is} ist die Menge, die alle Elemente enthält, die v_x klassifizieren.

Die Einzelschritte sind in Tabelle 9 aufgelistet.

Tabelle 9 Characteristics?-Algorithmus

1	Setze den Zeiger $\boldsymbol{v_x}$ auf das gesuchte Element. Wenn es sich bei $\boldsymbol{v_x}$ um ein Feature oder eine Data Source handelt, beende den Deduktionsprozess. $\boldsymbol{v_x \in L_3 \cup L_4 \rightarrow Ende}$
2	Füge jeden Positive-Constraint-Nachfolgerknoten von v_x zu S_{pos} hinzu. $\forall v_i \in N^+_{pos}(v_x) \mid S_{pos} \cup v_i$

3	Füge jeden Negative-Constraint-Nachfoglerknoten von v_x zu S_{neg} hinzu. $\forall v_k \in N^+_{neg}(v_x) \mid S_{neg} \cup v_k$
4	Füge jeden is-Nachfoglerknoten von v_x zu S_{is} hinzu. $\forall v_z \in N^+_{is}(v_x) \mid S_{is} \cup v_z$
5	Wenn es sich bei v_x um ein Item handelt, füge jeden has-Nachfolger von v_x zu S_{pos} hinzu. $v_x \in L_1 \rightarrow \forall v_n \in N^+_{has}(v_x) \mid S_{pos} \cup v_n$

Find!

Der derzeit umfangreichste und zugleich anwendungsstärkste Abstraktionsalgorithmus ist der *Find!*-Algorithmus. Dieser dient als Lösungsansatz der Problemstellung „Finde Elemente einer bestimmten Klasse zur Erfüllung eines gegebenen Zwecks!“. Zum Beispiel lautet eine mögliche Aufgabenstellung „Finde Software, um Anforderungen zu beschreiben!“. Dabei stellt „Software“ eine vorgeschriebene Klassifizierung der Lösungselemente dar. „Anforderungen beschreiben“ bildet das Ziel, in diesem Fall ein Combining aus Objekt und Activity, das mithilfe der Lösungselemente verfolgt werden soll.

Die konkreten Elemente, die gefunden werden sollen, werden in der Menge S abgelegt. Das abstrakte Element, welchem alle Elemente aus S entsprechen müssen, wird als v_a definiert (z.B. Software). Das Element, für das alle Elemente aus S genutzt werden können, wird als v_b definiert (z.B. Anforderungen beschreiben). Tabelle 10 enthält die erforderlichen Schritte bis zur Problemlösung. Unter der Annahme einer vollständigen Induktion verringert sich die Komplexität des zweiten Schrittes: Jeder Knoten, dessen Kind- und Kindeskindknoten gleiche used-for-Vorgänger aufweisen, besitzt selbst diese use-for-Vorgänger. Diese Schlussfolgerung ist in Abschnitt 4.2.2 (Seite 81) näher erläutert.

Tabelle 10 Find!-Algorithmus

1	Setze den Zeiger $\boldsymbol{v_x}$ auf $\boldsymbol{v_b}$. $\boldsymbol{v_x = v_b}$
2	Gehe auf die unterste part-of-Ebene von v_x. Die Schnittmengen aller used-for-Vorgänger der Elemente dieser Ebene bildet S_1. $\forall v_i \in N^-_{part_{max}}(v_x) \mid S_1 = N^-_{used}(v_i) \cup N^-_{used}(v_{i+1})$

2a	**Unter Annahme der vollständigen Induktion** Füge alle used-for-Vorgänger von v_x zu einer Menge S_1 hinzu. Weiter mit Schritt 3 $$S_1 = S_1 \cup N^-_{used}(v_x)$$
3	**S**etze den Zeiger v_x auf v_a: $$v_x = v_a$$
4	Füge alle is-Vorgänger von v_x zu einer Menge S_2 hinzu. $$S_2 = S_2 \cup N^-_{is}(v_x)$$
5	S ist die Schnittmenge aus S_1 und S_2. Beende den Deduktionsprozess. $$S = S_1 \cap S_2$$

4.2 Lernen und Gewichten

Gerade im Software Engineering unterliegt Wissen einem hohen Grad an Veränderbarkeit. Neue Erkenntnisse in Wissenschaft und Forschung, neuentwickelte oder erweiterte Softwarewerkzeuge und eine rasante Entwicklung der Hardwaretechnologie führen zu einer stark-dynamischen Wissensspeicherung. Daher ist eine der wichtigsten Funktionen des Modells, neues Wissen zu lernen und Schlussfolgerungen abzuleiten. Im Ontology Engineering stehen dafür verschiedene Methoden, wie das Ontology Learning oder Reasoner bereit (Abschnitt 2.3.5, Seite 36), an denen sich auch die hier vorgestellten Inferenzmechanismen orientieren.

Vorgestellt werden daher zwei Ansätze, um neues Wissen im Modell automatisiert erzeugen zu können. Zum einen wird ein dem Ontology Learning ähnlicher Ansatz dargelegt, der durch Analyse von natürlichen Sprachelementen neues Wissen in die Wissensbasis aufnehmen kann. Der zweite hier präsentierte Ansatz basiert auf induktiver Schlussfolgerung und kann Zusammenhänge zwischen bereits vorhandenen Wissenselementen aufdecken und automatisiert modellieren. Dazu dient unter anderem eine Untersuchung des Modells hinsichtlich künstlicher neuronaler Strukturen, um Mechanismen zur Mustererkennung in die Lerngestaltung einzubetten.

4.2.1 Lernen durch Pragmatik und Semantik

Ein wesentlicher Bestandteil der Wissensakquise bildet die Überführung von Wissen aus natürlicher Sprache in die formale Struktur des Modells. Diese Überführung kann einerseits manuell durch einen menschlichen Modellierer erfolgen, dem die Komponenten des

Modells mit all seinen Element- und Assoziationsklassen bekannt sind. Darüber hinaus besteht die Möglichkeit, dieses Wissen automatisiert aus Textbausteinen herauszulösen und in die Wissensbasis zu übertragen. Dafür müssen dem Automatisierungsmechanismus neben Überführungsalgorithmen, also Methoden zur korrekten Speicherung der Wissensartefakte, vor allem die Schlüsselbegriffe bekannt sein, aus denen sich Wissenszusammenhänge ergeben können. Diese Schlüsselbegriffe stehen in Abhängigkeit zur Textsprache (wie Deutsch, Englisch, Französisch, etc.). Daher ist beim Lernen durch Semantik und Pragmatik eine Aufteilung der Schlüsselbegriffe in verschiedene Sprachzentren unentbehrlich.

Das Lernen erfolgt durch Analyse der zu überführenden Text- oder Sprachartefakte. Diese Analyse wird durch eine Semantikerkennung durchgeführt, die auf Basis der bereits erwähnten Schlüsselbegriffe semantische Zusammenhänge zwischen den im Text enthaltenen Begriffen feststellt. Ein Algorithmus überführt anschließend diese Zusammenhänge und Begriffe in die Komponenten des Modells (Cells, Items, Feature, Activities und Combinings). Abb. 28 veranschaulicht dieses Vorgehen grafisch.

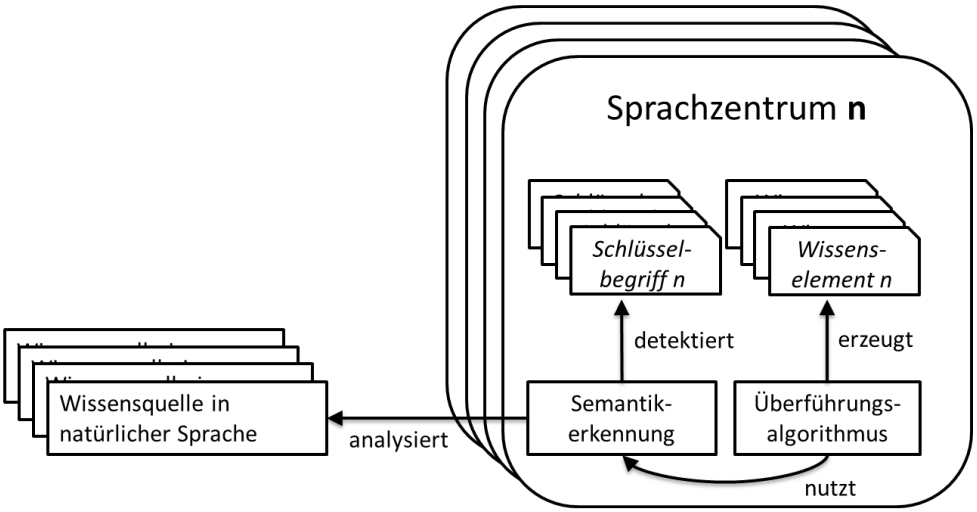


Abb. 28 Textbasiertes Lernen im Modell

An einigen englischsprachigen Beispielen wird im Folgenden das Vorgehen verdeutlicht. Die initialen Schlüsselbegriffe *a, each* und *all* lassen grundsätzlich auf etwas Allgemeines bzw. Klassifizierendes schließen. Im Sinne des Modells werden sie demnach gefolgt von einem Item. Dagegen weist der Schlüsselbegriff *the* auf etwas Konkretes, also eine Cell, hin. Subjekt-Objekt-Beziehungen werden durch die sich aus den Assoziationsklassen ergebenden Schlüsselwörtern *can, is, used for, has, must, should* oder *same as* deutlich.

Konkrete Anwendungsbeispiele zur Überführung von modellbasiertem Wissen aus natürlicher Sprache werden durch folgende Auflistung ersichtlich:

1. Each human can walk.
2. A person is a human.
3. Person is a human.
4. Franz is a person.
5. Füßl is the last name of Franz.

Im ersten Beispiel lässt das Schlüsselwort *each* darauf schließen, dass es sich bei *human* um eine abstrakte Klassen, also ein Item handelt. *can* hingegen weist auf die Activity *walk* hin. Zudem kann die Relation zwischen *human* und *walk* gebildet werden. Durch den zweiten Satz wird eine Vererbungsstruktur zwischen *person* und *human* deutlich. Durch den Schlüsselbegriff *a* wird auch *person* zu einem abstrakten Item. Im Gegensatz dazu würde *person* als konkrete Instanz von *human* modelliert werden, wenn Beispielsatz drei zugrunde gelegt wird. Im vierten Beispiel wird *franz* als Cell angelegt. Sollte zu diesem Zeitpunkt *person* auch eine Cell sein, wird diese automatisch zu einem Item geändert, da eine Instanz nicht die Klasse einer weiteren Instanz bilden kann. Das letzte Beispiel beinhaltet komplexere Wissenszusammenhänge. Es werden die Cell *füßl* und das Item *last name* erzeugt. Dabei bildet *füßl* eine Instanz von *last name*. Zusätzlich erfolgt die Verknüpfung von *franz* zu *füßl* durch eine Postive-Constraint. Dazu wird das Hinzufügen des boolschen Ausdrucks *r(franz) = r(füßl)* erforderlich. An dieser Stelle ist unklar, ob jede *person* einen *last name* besitzt. Da *franz* die Instanz einer *person* bildet, liegt dieser Rückschluss jedoch nahe. Daher könnte ein systemseitiges Hinterfragen zu einer has-Assoziation zwischen *person* und *last name* führen, wofür ein Anwender die Frage *Has each person a last name?* positiv beantworten müsste. Dieses Lernverhalten nach Nutzerrücksprache wird im nun anschließenden Abschnitt genauer beleuchtet.

4.2.2 Lernen durch induktive Schlussfolgerungen

Dieser Abschnitt betrachtet einige wesentliche Schlussfolgerungsregeln, die auf dem Zusammenhang zwischen Klassen und ihren Konkretisierungen basieren.

Merkmalsübertragung einer Abstraktion auf eine Konkretisierung

Etwas Konkretes, was durch eine is-Assoziation mit etwas Abstraktem in Beziehung gesetzt wurde, erbt die Eigenschaften des Abstrakten. Für den Fall, dass es sich beim erbenden Element um eine Cell handelt, kann unter Umständen nicht genau prognostiziert werden, welche konkrete Ausprägung der zu vererbenden abstrakten Eigenschaft auf das erbende Element zutrifft. Ein gängiger Ansatz, diesem Problem zu begegnen, ist der Einbezug von Gewichten, Wahrscheinlichkeiten und Interaktionen.

Jedes Item mit is-Assoziation zu einem anderen Item erbt die Eigenschaften des Elternelementes. Neben abstrakten Eigenschaften wie Form oder Farbe, können Items auch konkrete Merkmale aufweisen. Zum Beispiel besitzt jedes Getränk die Eigenschaft „flüssig“, was eine konkrete Instanz eines Aggregatzustands darstellt. In diesem Fall würde auch das konkrete Merkmal an alle abstrakten Getränk-Klassen vererbt werden.

Abb. 24 (Anwendungsbeispiele, Seite 65) verdeutlicht die Übertragung der abstrakten Eigenschaften *farbe* und *form* vom abstrakten Element *frucht* auf eine konkrete Instanz *apfel*. Diese Instanz bekommt konkrete Ausprägungen der Eigenschaften *farbe* und *form* durch Positive- und Negative-Constraints übertragen. Wird eine neue Instanz von *frucht* erzeugt, beispielsweise *birne*, kann durch die Übertragung der Eigenschaften erfragt werden: Ist die *birne* blau, rot oder grün und ist sie rund oder eckig?

Auch Feature können an Kindelemente vererbt werden. Werden sie als Eigenschaft eines abstrakten Objektes modelliert, stellen sie eine übertragbare Komponente zur Datenerfassung dar. Beispielsweise besitzt jede Software einen Preis, der erfasst werden muss, sobald eine neue Softwareinstanz in der Wissensbasis gebildet wird. Das item *Software* hat also die Eigenschaft *Preis*, welcher als Input-Feature modelliert ist. Dann wird beim Anlegen der neuen Cell *MS Word*, eine Dateneingabe zum *Preis* erforderlich. Diese Eingabe wird dann als Cell gespeichert und mit einer Positive-Constraint von *MS Word* assoziiert.

Merkmalsübertragung von Teilen auf ein Ganzes

Besteht ein Element aus mehreren Teilen, existiert also mindestens ein part-of-Vorgänger zu diesem Element, können alle Eigenschaften seiner Teile an das Element übertragen werden. Dieser Sachverhalt ist jedoch nur unter Einschränkungen gültig, da die Zusammenführung von Eigenschaften unter Umständen zu Konflikten führen kann. Beispielswei-

se besteht ein Notebook aus leitfähigen und nicht leitfähigen Bauteilen. Die automatisierte Übertragung dieser Eigenschaften auf das Ganze (Notebook) wäre in diesem Fall unzulässig. Daher ist eine genaue Untersuchung von Merkmalsübertragungen von Teilen auf ihr Ganzes an dieser Stelle zielführend, jedoch nicht Bestandteil dieser Arbeit.

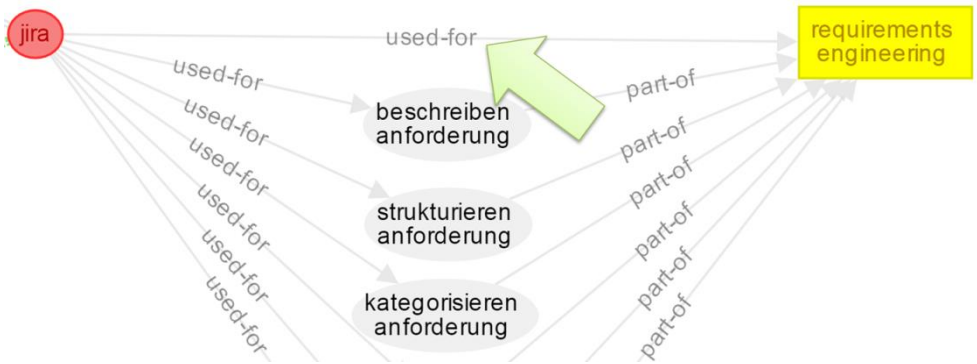


Abb. 29 Übertragung von Assoziationen

Ähnlich verhält es sich bei der Übertragung von Assoziationen seiner Teile auf ein Element. Nach Abb. 29 liegt die Annahme nahe, dass jede Cell und jedes Item mit part-of-Assoziationen dieselben Assoziationen übertragen bekommt, die alle seine Teile gemeinsam haben. Dies trifft im Falle des in der Grafik dargestellten *requirements engineering* mit seinen Teilaufgaben *beschreiben anforderungen*, *strukturieren anforderungen,* etc. und der Software *jira*, die zur Umsetzung dieser Teilaufgaben genutzt werden kann, zu. Eine Verallgemeinerung ist jedoch ohne Berücksichtigung verschiedener Restriktionen nicht möglich. Auch hier ist eine weiterführende Untersuchung der genauen Übertragungsmodalitäten hilfreich.

Erweiterung des Modells um Gewichte

Bei der Ausführung eines Konkretisierungsalgorithmus kann es zu Konflikte kommen. Beispielsweise kann die Eingabe eines Input-Features auf unterschiedliche Positive-Constraints verschiedener Elemente zutreffen. Diese Konflikte führen dazu, dass die Beantwortung einer Problemstellung nicht eindeutig vorgenommen werden kann, was unter Umständen Systementscheidung verhindert. Am Beispiel dreier Elemente mit identischem Positive-Constraint zum Feature *temperature* verdeutlicht Abb. 30 die Konfliktproblematik. Im Beispiel beträgt der erfasste Wert der Temperatur: *19*. Die Auswahl des Wetters erfolgte auf *sunny*. Diese Konstellation von Eingabeparametern führt zu einer Aktivierung der Elemente *favourite jacket* und *light jacket*. Eine eindeutige Entscheidung, welches

Element genau auf die Eingaben zutrifft ist demnach nicht möglich. Zudem ist beim Anlegen neuer Instanzen Konfliktpotenzial vorhanden, wenn vom Modell entschieden werden muss, welche konkreten Ausprägungen von Eigenschaften auf diese Instanzen zutreffen.

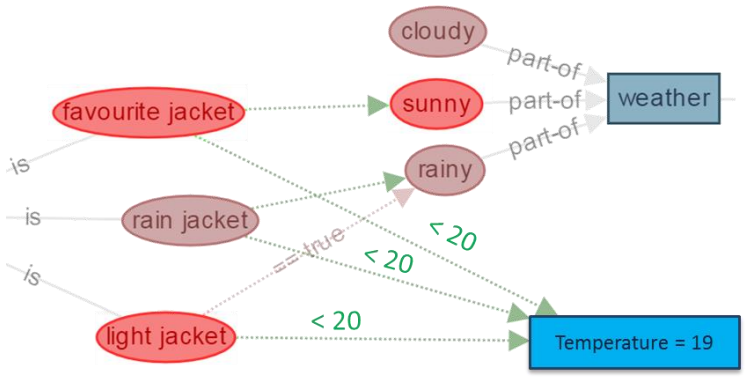


Abb. 30 Beispiel einer konfliktbehafteten Entscheidung

Eine Lösung des Problems liegt in der Integration von Gewichtungen und Wahrscheinlichkeiten, sowie in der Interaktion mit Nutzern. Gewichte dienen demnach dem Umgang mit Konflikten, um beim Lernen zu entscheiden, welche konkreten Ausprägungen einer Elterneigenschaft auf das Kind zutrifft. Auch kann das Durchlaufen der Deduktionspfade durch Priorisierung der Assoziationen effizienter gestaltet werden.

Die Gewichtung erfolgt indem jeder Kante ein Kantengewicht und jedem Knoten ein Knotengewicht zugewiesen wird, das einer reellen Zahl entspricht ($\omega : E \rightarrow \mathbb{R}$). Jeder Knoten und jede Kante des Graphen erhält eine Gewichtung in der Form:

$$\forall e \in E_G(v): \omega(e) \text{ und } \forall v \in V(G): \omega(v)$$

Dabei werden die Gewichte durch natürliche Zahlen vereinfacht: $\omega : E \rightarrow \mathbb{N}$. Kantengewichte setzen sich zusammen aus dem Grad eines adjazenten Knotens und einem vom System während des Lernens kalkulierten Faktor. Sollte eine Kante bereits eine höhere Gewichtung erhalten haben, behält sie ihr ursprüngliches Gewicht bei:

$$\forall e \in E_G(v): \omega(e) < d_G(v) \rightarrow \omega(e) = d_G(v) \; mit \; v \in V(G)$$

Trial and Error: Zufall, Wahrscheinlichkeiten und Nutzerinteraktion

Zur langfristigen Speicherung des auf Gewichtung basierten Wissens, stehen dem Modell zwei abstrakte Speicherzentren zur Verfügung: das Short-Term Memory (STM) zur kurz-

fristigen Speicherung von gelernten Gewichtungen und das Long-Term Memory (LTM), welches Elemente und Gewichtungen langfristig speichert. Das Verhalten der beiden Wissensspeicher orientiert sich an der menschlichen Wissensverarbeitung. Wissen wird langfristig im menschlichen Gehirn gespeichert, wenn damit eine Emotion verbunden ist, oder die Information oft genug wiederholt wurde.

Das STM enthält Elemente, die gänzlich neu oder bereits im LTM vorhanden sind und neue Gewichtungen erhalten haben. Jedes Element des STM besitzt einen Ablaufzeitpunkt und eine Anzahl, wie häufig das Element nach einer Nutzerentscheidung gewählt wurde. Das Ablaufdatum eines Elementes wird mit jeder Auswahl neu gesetzt. Nach siebenfacher Auswahl[5], erfolgt die Speicherung des Elementes und all seiner Assoziationen langfristig im LTM. Jedes Element des LTM hat einen Ablaufzeitpunkt, welches sich analog zum STM verhält, jedoch einen deutlich höheren Wert aufweist. Ein Element, dessen Ablaufzeitpunkt vergangen ist verliert seine Gewichte. Auch alle anliegenden Kanten verlieren ihre Gewichte. Der Inhalt des Elementes wird nicht gelöscht. Auf diese Weise wird die Priorisierung während der Deduktion unter einen zeitlichen Einfluss gestellt, ohne dass inhaltliches Wissen verloren geht. Somit wird maschinelles Lernen um maschinelles Vergessen ergänzt.

Ein beispielhafter Ablauf zur Wissensspeicherung in STM und LTM orientiert sich an den in Abb. 30 enthaltenen Elementen. Es wird davon ausgegangen, dass der verursachte Konflikt erstmalig auftritt. Nun stehen dem Modell drei verschiedene Möglichkeiten zur Verfügung, mit dem Problem umzugehen:

1. Das Modell wählt zufällig eine Jacke aus und fragt anschließend den Nutzer, ob die Auswahl korrekt war. Anschließend wird die Antwort des Nutzers in Form einer Gewichtung in die betreffenden Elemente integriert. Diese werden im STM abgelegt.
2. Das Modell verwendet zur Entscheidung, welche Jacke gewählt werden soll vormodellierte Wahrscheinlichkeiten, wonach im Konfliktfall beispielsweise *favourite*

[5] Eine Faustregel besagt, dass ein Mensch etwas im Durchschnitt siebenmal wiederholen muss, bis er es sich eingeprägt hat. Beispielsweise BECKER 2011; WADHWA 2005

jacket mit einer Wahrscheinlichkeit von 0.8 der korrekten Auswahl entspricht. Auch hier erfolgt im Anschluss an die Auswahl eine Nutzerrückfrage, deren Antwort die Wahrscheinlichkeiten ändert und im STM speichert.

3. Das Modell lässt den Anwender direkt selbst entscheiden und speichert die Auswahl per Gewichtung im STM.

Tritt dieser Konflikt erneut auf, wird das Vorgehen unter Berücksichtigung der im STM gespeicherten Informationen wiederholt. Nachdem siebenmal dasselbe Element ausgewählt wurde, werden alle mit dem Konflikt verbundenen Gewichte im LTM gespeichert. In zukünftigen Arbeiten kann untersucht werden, wann die Rückkopplung zum Anwender tatsächlich notwendig ist. Beispielsweise könnte nur jedes zweite oder dritte Mal eine Nutzerrückfrage gestellt werden, ohne die Prognosegenauigkeit zu beeinträchtigen.

Mustererkennung auf Feature- und Data Source-Ebene

Die letzte hier vorgestellte Möglichkeit zum automatisierten Lernen beschäftigt sich mit der Integration von Mustererkennungstechniken, die beispielsweise aus dem Gebiet der künstlichen neuronalen Netze bekannt sind. Im Rahmen dieser Arbeit erfolgt die Beschreibung des Ansatzes jedoch nur auf einem sehr modellhaften symbolischen Niveau, ohne vertiefte Techniken zur genauen Vorgehensweise darzulegen. Es soll lediglich die Vision aufgezeigt werden, die durch diesen Ansatz verfolgt wird.

Kern des Modells bildet die Verarbeitung von Wissen, welches über eine Eingabeschicht in der Wissensbasis erfasst wird. Diese Eingabeschicht besteht aus sogenannten Data-Source-Elementen, die durch Sensoren oder andere Messinstrumente als automatisierte Schnittstellen des Modells Daten erfassen können. Die Überführung dieser Daten in Informationen und Wissen wird anschließend durch Feature und Cells im Sinne des Modells vorgenommen (vgl. Abschnitt 3.2, Seite 44). Bei bisherigen Versuchen und Prototypen wurde die Datenerfassung manuell durch Nutzeraktivitäten vorgenommen. Sie kann jedoch auch vollautomatisiert während der Deduktion als Datenstrom in die Entscheidungsprozesse einfließen. Wird dieser Datenstrom um Mustererkennungstechniken ergänzt, kann dies zu einer automatischen Überführung von Daten zu Wissen führen.

Abb. 31 stellt das bereits gezeigt Beispiel der automatischen Jackenwahl in den Fokus dieses Lernmechanismus. Auf der rechten Seite der Grafik sind Sensoren abgebildet, deren erfasste Daten in verschiedene Feature einer Ontologie übertragen werden. Welche Kom-

bination der Daten dabei zu welcher Wissenszelle führt, wird der Ontologie vom Nutzer durch Interaktion beigebracht. Die Ergebnisse werden durch Verändern oder Hinzufügen von Positive- und Negative-Constraints der vom Nutzer zugeordneten Elemente in der Wissensbasis gespeichert. Dieser Prozess wird solange wiederholt, bis die Treffer des Systems akzeptabel sind.

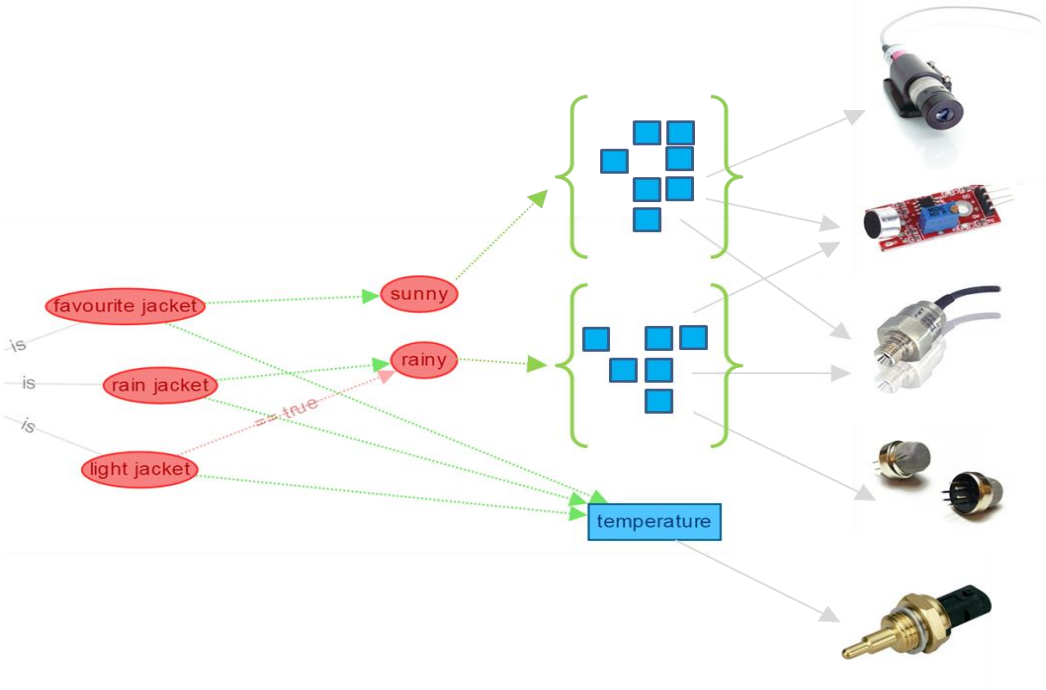


Abb. 31 Integration von Mustererkennung bei großen Datenmengen

Dabei dienen Methoden und Verfahrensweisen aus dem Bereich der künstlichen neuronalen Netze als Orientierungshilfe. Sinngemäß können die Ebenen der Feature und Data-Source-Elemente als künstliches Neuronales Netze betrachtet werden, bei dem die einzelnen Elemente den Neuronen des Netzes entsprechen. Das Erfassen von Daten entspricht der bei mehrschichtigen feedforward-Netzen typischen Eingabeschicht. Auf diese Weise können bekannte Ansätze des maschinellen Lernens in das Modell integriert werden, wie Supervised Learning, Unsupervised Learning oder Reinforcement Learning. Auf weiterführende Erläuterungen zu den einzelnen Ansätzen wird im Rahmen dieser Arbeit verzichtet.

5 Evaluierung

5.1 Prototypische Umsetzung des Modells als Webanwendung

Die Evaluierung des Modells erfolgt in zwei Schritten. Zur Durchführung beider Schritte wird ein webbasierter Prototyp entwickelt, der alle erforderlichen Modellkomponenten und die beiden komplexesten Deduktionsalgorithmen „Isn't-It?" und „Find!" unterstützt. Die Algorithmen „Parts?", „Characteristics?" und „Kind-Of?" können im Rahmen weiterführender Arbeiten unkompliziert ergänzt werden. Dieser Abschnitt dokumentiert Anforderungen, Architektur und Implementierung, einschließlich einer Gegenüberstellung von Soll- und Ist-Verhalten des Prototyps durch ausgewählte Testfälle.

5.1.1 Anforderungsbeschreibung

Trotz prototypischer Umsetzung unterliegt die zu entwickelnde Software einiger funktionaler und nicht-funktionaler Anforderungen. Auf die Auflistung von Qualitätsmerkmalen wird in diesem Zusammenhang verzichtet. Im Folgenden werden die Anforderungen an den Prototyp aufgelistet und genauer beschrieben:

A01 – Elementklassen anlegen: Der Prototyp muss die Möglichkeit bieten, Items, Cells, Features, Combinings, Activities und Questions anzulegen und grafisch zu visualisieren. Die Visualisierung der Elemente erfolgt durch eine farbliche Differenzierung: Cells sind rote Ellipsen, Items sind gelbe Rechtecke, Feature sind blaue Rechtecke, Activities sind weiße Sechsecke, Combinings sind graue Ellipsen und Questions sind lilafarbene Rauten. Beim Anlegen wird jedem Element eine eindeutige ID zugewiesen. Jedes Element erhält einen Namen (Bezeichnung), der aus beliebig vielen alphanumerischen Zeichen besteht und bei Questions die genaue Fragestellung darstellt. Alle Sonderzeichen sind zugelassen. Im Falle einer Cell kann zusätzlich ein boolescher Ausdruck angegeben werden, der das Ergebnis der Cell bestimmt. Beim Anlegen eines Features soll der Nutzer zwischen den Featuretypen Single-Choice, Multiple-Choice oder Input-Feature wählen können. Ein

Combining besteht aus einer Activity oder einer Cell und einem Objekt (Cell oder Item). Beide Elementtypen können vom Nutzer bestimmt werden.

A02 – Assoziationsklassen anlegen: Es muss möglich sein, die verschiedenen Basis-Assoziationstypen used-for, part-of, can, is und has anzulegen, sowie die Abhängigkeitsbeziehung „Negative-Constraint“. Jede Kante erhält eine eindeutige ID. Zusätzlich ist eine Unterscheidung nach optionaler und verpflichtender Kante konfigurierbar zu gestalten. Die Assoziationen müssen visualisierbar sein, indem beim Erzeugen der Kante der Klassennamen auf die Kante geschrieben wird. Negative-Constraints sind rot gepunktet. Beim Anlegen einer Assoziation muss der Quellknoten und der Zielknoten bestimmt werden können. Wird eine can-Assoziation erzeugt, kann zusätzlich eine Activity angeben werden, die in Verbindung mit dem Zielknoten zu einem Combining führt. Beim Anlegen eines Negative-Constraints muss zusätzlich die Eingabe eines booleschen Ausdrucks ermöglicht werden, der sichtbar auf der Kante platziert wird.

A03 – Benutzeroberfläche: Für den Prototyp muss eine Benutzeroberfläche zur Verfügung stehen, bei der die Wissensartefakte (Element- und Assoziationsklassen) visuell dargestellt werden. Ein Verschieben dieser Wissensartefakte soll per Drag’n’Drop ermöglicht werden. Eine Selektion von Knoten oder Kanten ist möglich. Wird ein Element selektiert kann es durch einen Buttonklick oder durch Drücken der Taste *Entf* gelöscht werden. Drückt der Nutzer die Taste *n* öffnet sich das „Elementklasse anlegen“-Menü. Bei Betätigen der Taste *e* öffnet sich das „Assoziationsklasse anlegen“-Menü. Der Nutzer hat die Möglichkeit den Darstellungsbereich im Vollbild anzeigen zu lassen. Durch Betätigen des Mausrades kann der Anwender die Ontologie vergrößern bzw. verkleinern (zoomen). Durch Klicken und Halten der linken Maustaste kann der Nutzer den Fensterausschnitt verschieben.

A04 – Ontologie laden/speichern: Der Nutzer hat die Möglichkeit, eine von ihm erstellte Ontologie in einem Verzeichnis seiner Wahl zu speichern oder von einem beliebigen Speicherort zu laden. Dazu existieren jeweils Buttons, die bei Klicken ein entsprechendes Abfragefenster aufrufen. Beim Speichern wird der Nutzer aufgefordert einen Namen (keine Einschränkungen) für die Ontologie anzugeben.

A05 – Elementklasse bearbeiten: Hat ein Nutzer eine Elementklasse selektiert, besteht die Möglichkeit ein „Bearbeiten“-Button zu betätigen. Es öffnet sich ein Dialog, in dem er

die Klasse des Elementes und den Namen ändern kann. Für den Fall, dass es sich bei dem selektierten Element um eine Cell handelt, kann er zusätzlich den booleschen Ausdruck zur Erzeugung des Ergebnisses ändern. Möchte der Nutzer ein Element zu einem Combining ändern, ist er gezwungen die erforderlichen Objekte (Activity, Cell, Item) auszuwählen (siehe A01). Zusätzlich wird im „Bearbeiten"-Dialog die ID des Knoten angezeigt. Beim Bearbeiten eines Features kann der Featuretyp geändert werden.

A06 – Ergebnisausdrücke: Ergebnisse einer Cell werden in Form von booleschen Ausdrücken deklariert. Um das Ergebnis eines anderen Knoten in den Ausdruck zu integrieren wird die ID des anderen Knoten von runden Klammern umschlossen und durch ein kleines *r* eingeleitet (Beispiel: *r(knoten_id)*). Der Nutzer hat die Möglichkeit, die Vergleichsoperatoren ==, &&, ||, >, <, >=, <= in den Ausdruck einzubeziehen und den Ausdruck durch runde Klammern zu verschachteln. Schließt der Nutzer durch Bestätigen des „Anlegen"- oder „Bearbeiten"-Dialoges die Eingabe ab, wird vom Prototyp eine Verknüpfung zu den im Ausdruck enthaltenen Knoten erzeugt. Die Verknüpfung wird als Positive-Constraints (grüne gepunktete Linie) dargestellt.

A07 – Deduktionsmöglichkeit: Der Anwender hat die Möglichkeit verschiedene Deduktionsalgorithmen auf die von ihm geladene Ontologie anzuwenden. Der Prototyp implementiert die komplexen Algorithmen „Isn't-It?" und „Find!" und bietet darüber hinaus für alle vorgestellten Algorithmen die architektonische Grundlage, um die trivialeren Algorithmen (Kind-Of?, Parts? und Characteristics?) jederzeit ergänzen zu können. Für jeden Algorithmus existiert ein Button, der bei Klick einen Dialog öffnet, in dem der Nutzer weitere Informationen angeben muss. Beim Klick auf den Isn't-It?-Button öffnet sich ein Dialog zur Auswahl des Knotens, der geprüft werden soll. Dazu gibt der Nutzer den Namen des gewünschten Knotens in ein Textfeld ein. Ein Autocomplete stellt passende Ergebnisse dar, aus denen der Nutzer den gewünschten Zielknoten auswählt. Nachdem er einen Knoten gewählt hat, beginnt der Prototyp mit der Abarbeitung des Algorithmus. Sind während der Deduktion Nutzereingaben erforderlich, werden diese auf dem Bildschirm in Form von Dialogfenstern ausgegeben. Single-Choice-Feature werden durch Radiobuttons, Multiple-Choice-Features durch Checkboxen und Input-Feature durch Textfelder dargestellt. Am Ende der Deduktion erscheint das Ergebnis auf dem Bildschirm durch eine Textausgabe: *Es handelt sich um eine/n $ELEMENTNAMEN* bzw. *Es handelt sich nicht um eine/n $ELEMENTNAMEN*. Dabei ist *$ELEMENTNAMEN* durch den Namen des Elementes zu

ersetzen, über das deduziert wurde. Beim Find!-Algorithmus muss der Nutzer zwei Informationen angeben: welche Abstraktion gesucht wird und für was das gesuchte Element genutzt werden soll. Ergebnis des Find!-Algorithmus ist eine Auflistung der Lösungsmenge.

A08 – Webbasiert: Der Prototyp soll webbasiert als Browseranwendung genutzt werden können und auf JavaScript, PHP, CSS und HTML basieren. Jede Form von Bibliotheken ist zulässig. Eine Nutzerauthentisierung oder Verschlüsselung ist nicht notwendig.

5.1.2 Architektur und Implementierung

Den Kern des Prototyps bilden aufgrund der Anforderung A08 PHP- und JavaScript-Komponenten. Die Darstellung der GUI erfolgt durch JavaScript, HTML und CSS. Die Basisarchitektur wird von dem PHP-Framework „Yii“ in Version 1.1 getragen. Es basiert auf einer Model-View-Controller (MVC) Architektur und bietet neben seiner sehr schlanken Struktur und seines modularen Aufbaus einen hohen Grad an Effizienz und Erweiterbarkeit. Yii wird mit dem Open Source CMS Wordpress kombiniert, um die Seitenstruktur und das Templatedesign der Anwendung für Administratoren zu erleichtern. Zusätzlich werden die JavaScript-Bibliotheken jQuery (Version 1.11) für verschiedene GUI-Interaktion und cytoscape.js zur Integration von Graph-Elementen (Knoten und Kanten) verwendet. Abb. 32 zeigt die einzelnen Komponenten des Prototyen und ihre Zusammenhänge.

Über den Web Browser werden durch HTML Formulare, die in den Wordpress Template-Pages enthalten sind, mittels jQuery Nutzereingaben erfasst. cytoscape.js ist eine Bibliothek zur grafischen Darstellung und Interaktion mit Knoten und Kanten eines Graphen. Da das hier erarbeitete Modell als Graph betrachtet werden kann, wird cytoscape.js zur Visualisierung der Modellklassen (Cell, Feature, Item, etc.) und damit zur Visualisierung des modellierten Wissens herangezogen. Zusammen bilden sie die View des MVC-Prinzips. Wissen, was auf diese Weise vom System erfasst wird, wird von internen JavaScript Klassen in eine einheitliche Struktur gebracht und somit digital überführt. Dieses überführte Wissen wird anschließend an Yii übermittelt und vom Controller auf dem Web Server verarbeitet. Die Weitergabe der Daten erfolgt via AJAX in Form von GET und POST Requests. Das Format zum Austausch und zur Speicherung von Daten ist JSON.

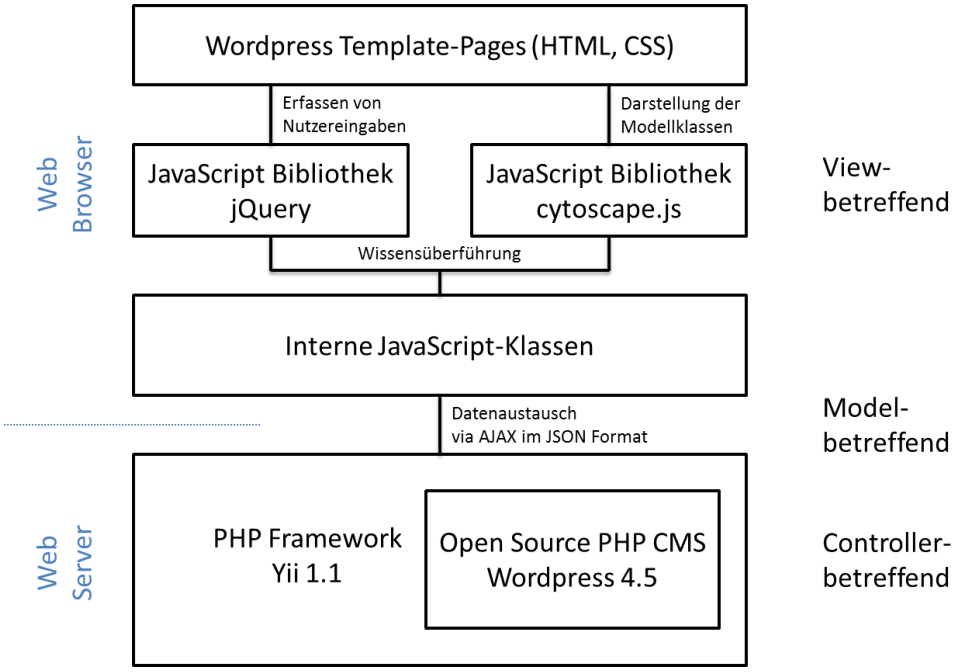


Abb. 32 Grundarchitektur des Prototypen

Architekturübersicht

Basierend auf der internen MVC-Architektur des Yii-Frameworks ergeben sich für den Prototyp die in Abb. 33 dargestellten Klassen und Komponenten. Die Controller-Klassen enthalten logische Komponenten des Prototyps, die meist in Form sogenannter Actions implementiert werden. Diese Actions können aus dem View heraus per URL aufgerufen werden. Zudem initiieren die Controller das Laden einer View und die Verarbeitung von Daten durch angebundene Modells. Der *KnowbaseController* lädt den Mainview der *knowbase*-Viewkomponente und sorgt für das Speichern des überführten Wissens durch Erzeugen einer JSON-Datei. Der *DeductionController* lädt den Mainview der *deduction*-Viewkomponente. Er enthält die Actions, die zur Ausführung der Deduktionsalgorithmen notwendig sind (bspw. Find!-Algorithmus). Zusätzlich enthält er die Logik zur Verarbeitung von Feature-Daten, die durch Nutzereingaben entgegen genommen werden. Der *WordpressController* sorgt schließlich für das Laden des Wordpress-Grundgerüstes. Er leitet jede Anfrage an die Wordpresskomponenten weiter, die nicht dem *Knowbase* oder *DeductionController* zugeordnet werden kann.

Die Model-Ebene (im Sinne des MVC) wird maßgeblich durch eine dem Strategy Pattern entsprechende Architektur aufgebaut. Konkrete Strategien bilden die Klassen *Item, Cell, Feature, Combining, Activity* und *Question*. Diese erben von der abstrakten Klasse *Node*.

Ziel ist einheitliche Methodenaufrufe zur Ergebniserzeugung und zur Suche relevanter Knoten auf einem Deduktionspfad zu erhalten. Jeder Knoten (*Node)* kennt seine Vorgänger und Nachfolger, sowie die jeweiligen Assoziationsklassen, mit denen er mit seinen adjazenten Knoten verbunden ist. Die Klasse *Graph* beinhaltet alle in der Zielontologie enthaltenen (Wissens)-Knoten und Assozationen *(Edges).*

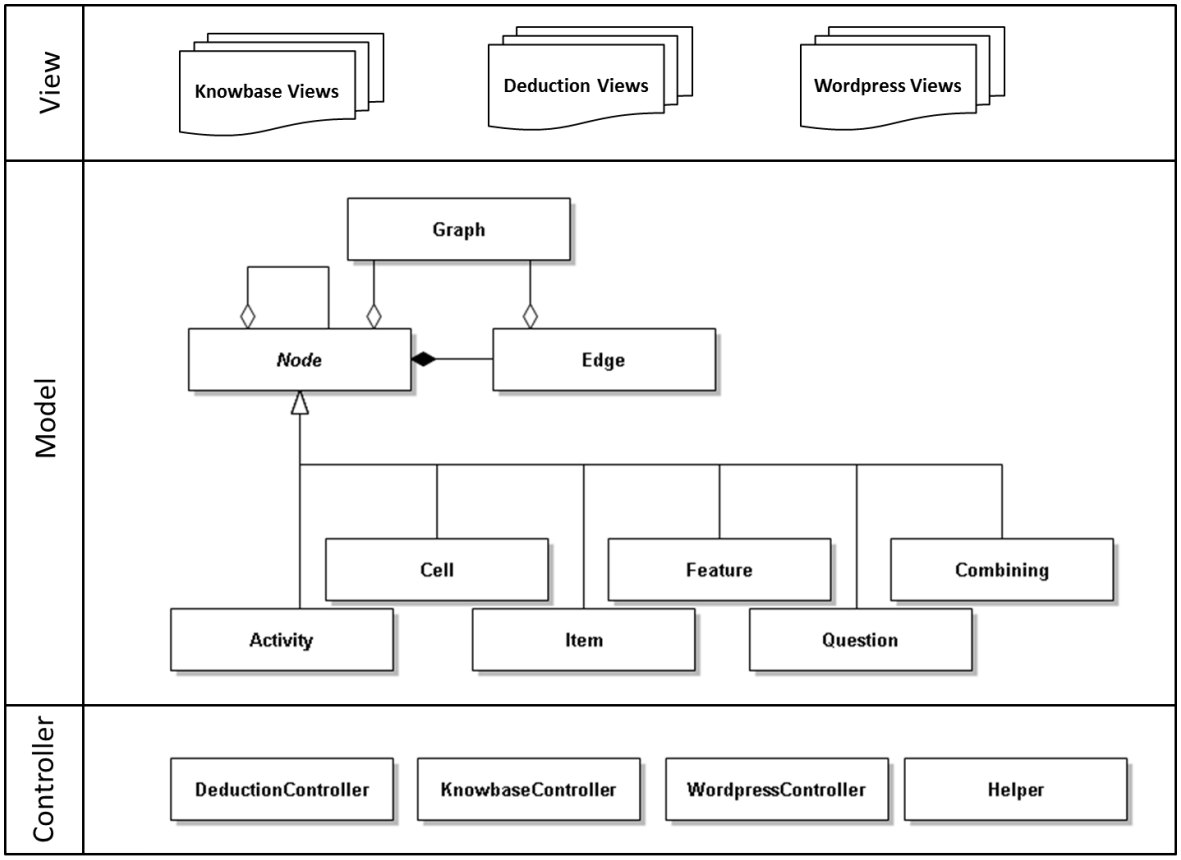


Abb. 33 Detaillierte Sicht auf die MVC Architektur

Eine besondere Rolle spielt die statische *Helper*-Klasse. Sie wird im Rahmen des Yii-Frameworks als Komponente geladen und enthält statische Methoden, die von jeder Klasse des Prototyps genutzt werden können. Beispielsweise enthält *Helper* eine vom Factory Pattern abgeleitete Funktion zur Initialisierung konkreter Model-Strategien (Cell, Item, Feature, etc.).

Datenmodell im JSON Format

Zur Speicherung und zum Austausch von Wissenselementen (Cell, Item, etc.) unterstützt der Prototyp das JSON (Java Script Object Notation) Format. Dieses Format dient zum standardisierten Austausch von Daten in strukturierter Form. Ähnlich der XML bietet

JSON die Definition eigener Bezeichner und eine Methode diese mit Daten zu befüllen. Die für den Prototyp verwendete Struktur entspricht der aus Abb. 34. Jedes Datenobjekt besitzt Felder, die zur Visualisierung des Graphen notwendig sind (grün dargestellt) und Felder, die per AJAX an die serverseitige Verarbeitung in Controller und Models versendet werden (blau dargestellt). Die grau markierten Objekte sind verschachtelte Objekte, die ihrerseits eigene Attribute enthalten. Hybride Daten sind Daten, die sowohl zur Anzeige, als auch zur serverseitigen Datenverarbeitung dienen. Diese werden als lilafarbene Ellipsen visualisiert. *edge* und *node* sind hier nur als abstrakte Objekte zu betrachten, die in der JSON-Datei nicht explizit formuliert werden.

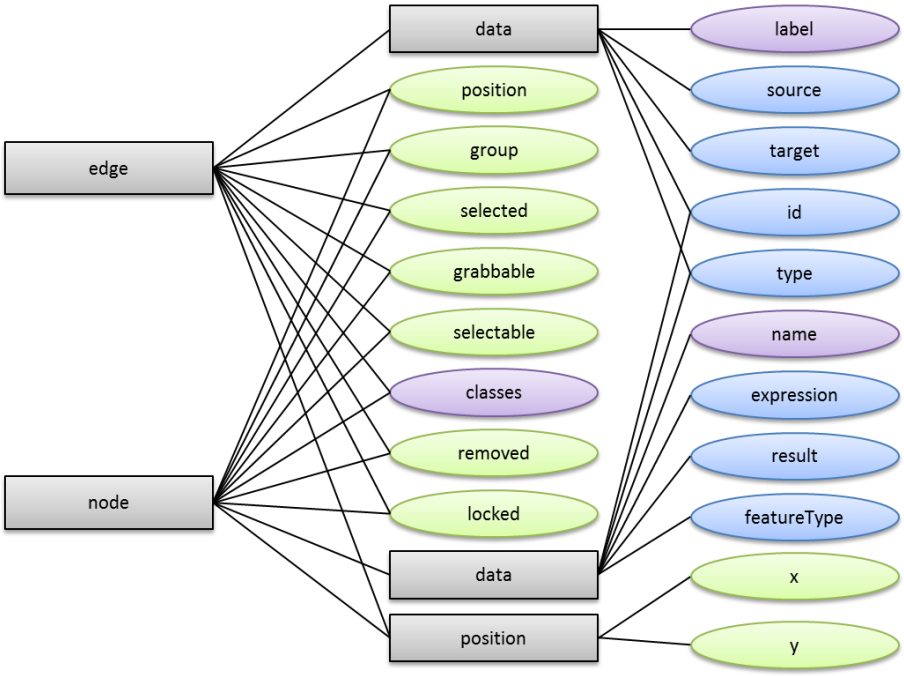


Abb. 34 Datenmodell der beiden JSON Klassen edge und node

Einen Auszug aus einer im Prototyp entstanden JSON-Datei ist Abb. 35 zu entnehmen. Darin enthalten sind ein *node*-Objekt und ein *edge*-Objekt mit ihren jeweiligen Attributen. Wie den Anforderungen zu entnehmen ist, soll jedes Element mittels GUID eindeutig identifiziert werden. Zudem bietet ein Knotenelement die Möglichkeit, einen Namen, einen Typ, einen booleschen Ausdruck, ein Ergebnis und für den Fall, dass es sich dabei um ein Feature handelt, den entsprechenden Feature Typ zu bestimmen. Im unteren Teil der Abbildung sind die Datenfelder einer Kante aufgelistet. Im Unterschied zum Knotenobjekt werden hier eine *source* (Quellknoten) und ein *target* (Zielknoten) gespeichert. Dabei die-

nen die IDs der Knoten als Referenzattribut. Zusätzliche Felder sind der entsprechende Assoziationstyp (hier *part-of*) und ein Kantenlabel.

```
[{
      "data": {
         "id": "b86c0bc2-a62b-1e93-39c1-4963528a937e",
         "name": "anforderungsmanagement",
         "type": "item",
         "expression": "r(97f2e566-37cf-778e-98b5-b6cfaf2bcad7)>=2",
         "result": null,
         "featureType": null
      },
      "position": {
         "x": -773.3659228317438,
         "y": -403.17072682176047
      },
      "group": "nodes", "removed": false, "selected": false, "selectable": true,
      "locked": false, "grabbable": true, "classes": "item"
}, … ,{
      "data": {
         "id": "5a6f480c-23da-4e4a-5f2b-e06513383ba4",
         "source": "923e4129-8201-7b81-b6b3-97806d5d65f2",
         "target": "dbd1877f-5905-e088-99b9-80f45b7371c0",
         "type": "part-of",
         "label": "part-of"
      },
      "position": {},
      "group": "edges", "removed": false, "selected": false, "selectable": true,
      "locked": false, "grabbable": true, "classes": "required"
}, … ,
]
```

Abb. 35 Auszug aus einer JSON-Datei mit einem Knoten und einer Kante

Zur Positionierung und Darstellung eines Elementes auf der Benutzeroberfläche stehen das *position*-Objekt und die Felder *group, removed, selected, selectable, locked, grabbable* und *classes* zur Verfügung. Diese Felder sind erforderlich, um den Graphen mithilfe der cytoscape.js darzustellen.

Detaillierte Sicht auf die Klassen des Prototyps

Abb. 36 zeigt die Model- und Controller-Klassen, ihre Methoden und Membervariablen sowie die Beziehungen zwischen den einzelnen Elementen. Die Klasse *Graph* wird beim Aufruf einer Deduktions-Action im *DeductionController* initialisiert. Sie enthält vier Arrays, welche den in Abschnitt 3.2.1 (Seite 46) vorgestellten Mengen E, V, M und P entsprechen. Der *Graph* Konstruktor wird unter Angabe eines Pfades aufgerufen. Dieser gibt den relativen Pfad zur Datei an, welche die zu ladenden Knoten und Kanten enthält. Das

Importieren der Knoten und Kanten aus dieser Datei erfolgt in der privaten Methode *importFile()*, welche vom Klassenkonstruktor aufgerufen wird.

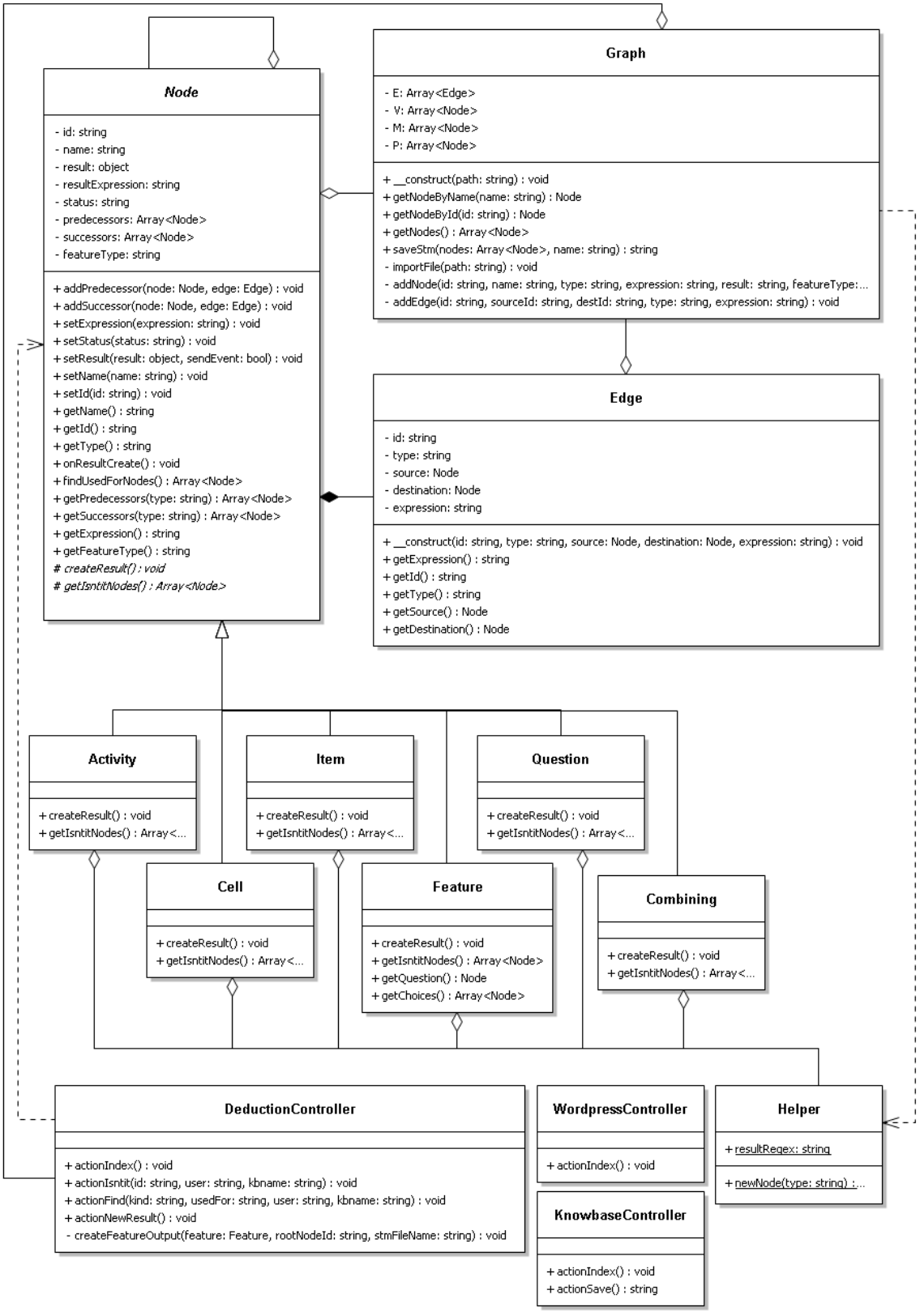


Abb. 36 Klassendiagramm für Graph, Node und Edge

Die Methoden *getNodeByName(string name)* und *getNodeById(string id)* geben jeweils einen Knoten zurück, der zu den gesuchten Attributen passt. Hingegen werden von *getNodes()* alle Knoten der Menge *V* zurückgegeben. Mittels *saveStm(Array<Node> nodes, string name)* werden Zwischenergebnisse während der Deduktion in einer JSON-Datei abgespeichert. Diese Funktion wird nach jedem vom Nutzer neu erfassten Feature-Ergebnis, also vom *DeductionController* aufgerufen. Die Funktionen *addNode* bzw. *addEdge* werden beim Importieren der Wissenselemente aus einer Datei aufgerufen. Sie erzeugen *Node*- und *Edge*-Objekte aus dem im JSON-Format vorliegenden strukturierten Wissen. Beim Erzeugen der konkreten Elementklassen (Cell, Item, usw.) bedient sich die Funktion der Factory Initialisierung aus der *Helper*-Methode *newNode*, die als Parameter den Typ des Elements übergeben bekommt und ein entsprechendes Objekt zurückgibt (Abb. 37):

```
6   public static function newNode($type) {
7       $node = null;
8       if ($type == "item")
9           $node = new Item();
10      else if ($type == "cell")
11          $node = new Cell();
12      else if ($type == "feature")
13          $node = new Feature();
14      else if ($type == "question")
15          $node = new Question();
16      else if ($type == "combining")
17          $node = new Combining();
18      else if ($type == "activity")
19          $node = new Activity();
20      return $node;
21  }
```

Abb. 37 Umsetzung des Factory Pattern in der Klasse „Helper“

Das von der Funktion *newNode()* zurückgegebene Objekt entspricht einer konkreten *Node*-Strategie. Der Grund für die Anwendung des Strategy Pattern bezüglich der Elementklassen des Modells liegt in der Erzeugung der Ergebnisse, die, wie Abschnitt 3.4.2 (Seite 64) zeigt, von Elementklasse zu Elementklasse abweichen kann. Die Strategien sind von der abstrakten Klasse *Node* abgeleitet. Jedes dieser Objekte enthält einen Namen, eine ID, gegebenenfalls einen booleschen Ausdruck zur Erzeugung des Ergebnisses, das Ergebnis selbst, einen Status und alle seine Nachfolger und Vorgänger. Zur Initialisierung dieser Membervariablen existieren jeweils eigene Getter- und Setter-Methoden. Eine Besonderheit bildet die *Node*-Methode *onResultCreate()*, die als Event verstanden werden kann. Sobald ein Knoten sein Ergebnis erzeugt hat, wird von *onResultCreate()* veranlasst, dass bei allen Observern des Objektes, also bei allen Negative-Constraint- und Postive-

Constraint-Vorgängern, sowie bei allen is-Nachfolger-Abstraktionen, das eigene Ergebnis erzeugt werden kann.

Implementierung des Deduktionsalgorithmus: Isn't-It?

Die beiden Deduktionsalgorithmen Isn't-It? und Find! sollen vom Prototyp unterstützt werden (Anforderung A07). Da es sich bei dem Prototypen um eine Webanwendung handelt, ist die Implementierung der Algorithmen insbesondere des Isn't-It?-Algorithmus komplizierter, als beispielsweise bei einer Windowsanwendung. Der Grund für die erhöhte Komplexität liegt im Request/Response Prinzip zwischen serverseitigem und clientseitigem Code. Zur Laufzeit der Anwendung erfolgt das Erfassen von Nutzereingaben per JavaScript und HTML über den Browser des Anwenders. Diese Eingaben werden dann asynchron an den Server per URL-Requests weitergeleitet und dort verarbeitet. Bei mehreren aufeinander aufbauenden Nutzereingaben, wie das beim Isn't-It?-Algorithmus der Fall ist, müssen vorherige Eingaben zwischengespeichert werden und können nicht in einem Workflow verarbeitet werden.

In Form eines Sequenzdiagramms enthält Abb. 38 die Laufzeitsicht auf die betroffenen Komponenten, die während des Isn't-It?-Algorithmus aktiv sind. Diese Komponenten sind die Deduction-View *index*, der *DeductionController*, die *Helper*-Klasse und die Models *Graph, Node, Edge, Feature* und *Question*. Konkrete *Node*-Strategien wie *Cell* oder *Item* werden in der Grafik nicht berücksichtigt. Der Einstieg erfolgt über die View *index*. Diese ruft nach einem Button-Klick per AJAX die URL

/deduction/isntit/*startknoten_id*/*benutzer_name*/*ontology_name*

auf. Diese URL wird vom Yii-Framework an die Action *actionIsntit()* des *DeductionControllers* weitergeleitet. Sie beinhaltet das Initiieren eines *Graph*-Objektes durch Aufrufen des entsprechenden Konstruktors. Dabei wird der Pfad zur JSON-Datei übergeben, welche die modellierten Wissenselemente enthält.

Im *Graph*-Konstruktor erfolgt durch *importFile()* die Überführung der textbasierten Elemente in Objekte. Dazu dienen die Methoden *addNode()* und *addEdge()*. Durch Implementierung des Factory Pattern wird beim Aufruf von *newNode()* eine konkrete *Node*-Strategie an die *addNode()*-Methode zurückgegeben und in *V* gespeichert. Nachdem alle in der zu überführenden Datei enthaltenen Wissenselemente in *V* und *E* des *Graph*-Objekts gespei-

chert wurden, ruft *actionIsntit()* den zu deduzierenden Startknoten aus dem *Graph*-Objekt auf und lässt sich alle erforderlichen Knoten geben, die auf dem Deduktionspfad liegen.

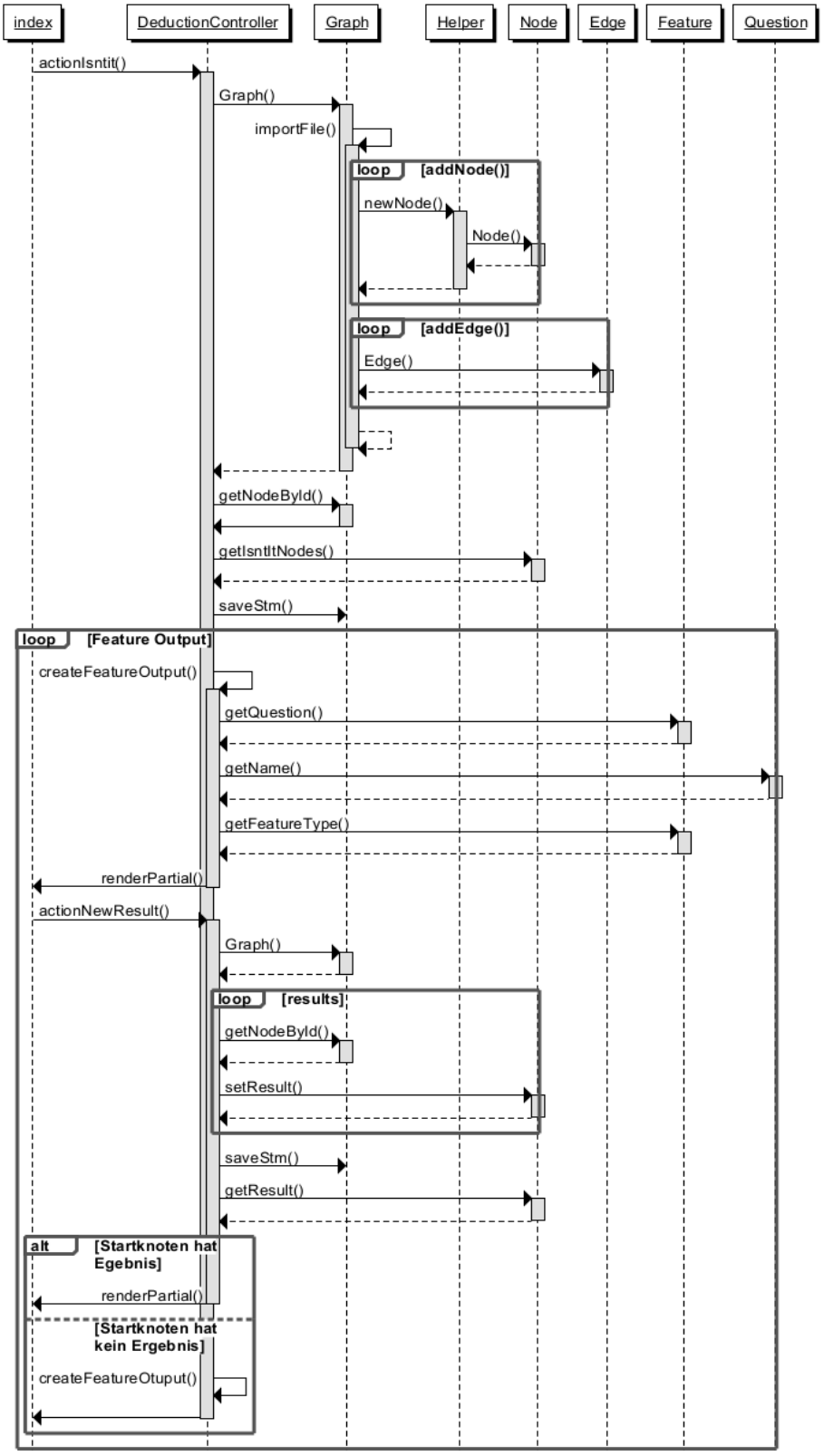


Abb. 38 Sequenzdiagramm zu Isn‘t It?-Algorithmus

Die Wahl, welche Knoten für die Isn't-It?-Deduktion erforderlich sind, bestimmt jeder Knoten in Abhängigkeit seiner Elementklasse selbst. Anschließend werden diese Elemente in einer eigenen Datei durch *saveStm()* unter einem eindeutigen Namen gespeichert. Da der Isn't-It?-Algorithmus auf Eingaben vom Nutzer angewiesen sein kann und diese Eingaben über Feature erfasst werden, sucht der Algorithmus nun nach dem ersten Feature ohne Ergebnis und gibt dieses über den Browser des Anwenders aus. Dazu übergibt er an die View die Fragestellung und den Feature-Type (multiplie-choice, single-choice oder input) zur automatisierten Auswahl einer entsprechenden Teilview. Der Aufruf der View erfolgt über *renderPartial()*, eine im Yii-Framework enthaltene Core-Funktionen, die jeder Controller erbt, der von *CController* abgeleitet ist.

Nachdem die Eingabe eines Nutzers erfasst wurde, wird diese über die Action *actionNewResult()* an den *DeductionController* per POST-Request übermittelt. Die Action erzeugt daraufhin wieder ein *Graph*-Objekt aus der Datei, deren Namen an die Action übergeben wurde. Im Anschluss wird jedes übermittelte Ergebnis, das aus der Nutzereingabe resultiert, durch *setResult()* des entsprechenden Knotens gesetzt. Sollte es sich bei der erfassten Eingabe um ein multipl-choice- oder singel-choice-Feature handeln, werden mehrere Ergebnisse übertragen, da diese Feature von allen angebundenen Auswahlmöglichkeiten (Cells) die Ergebnisse bilden können. Daher erfolgt der Aufruf der *setResult()*-Methode in einer Schleife. Wie bereits beschrieben, ruft ein *Node*-Objekt bzw. eine konkrete *Node*-Strategie bei all seinen Observern die Methode *createResult()* auf, sobald dieses Objekt sein Ergebnis gesetzt (bekommen) hat. Auf diese Weise erfolgt die Abarbeitung der Deduktionsknoten bis das Ergebnis des Startknoten gefunden wurde.

Nach einem erneuten Speichern des Graphen wird geprüft, ob das Ergebnis des Startknotens vorhanden ist. Sollte dies der Fall sein, wird sein Ergebnis an die View übergeben und der Algorithmus ist an dieser Stelle beendet. Kann kein Ergebnis vom Startknoten ermittelt werden, erfolgt die Ausgabe des nächsten „ergebnislosen" Features und der Teilprozess „Feature Output" beginnt erneut.

Implementierung des Deduktionsalgorithmus: Find!

Der Find!-Algorithmus dient zur Suche nach einem Element, welches einer bestimmten Klasse entspricht und zur Lösung einer konkreten Aufgabe genutzt werden soll. Abschnitt 4.1.2 (Seite 76) beschreibt die Zielstellung und das Verhalten des Algorithmus genauer.

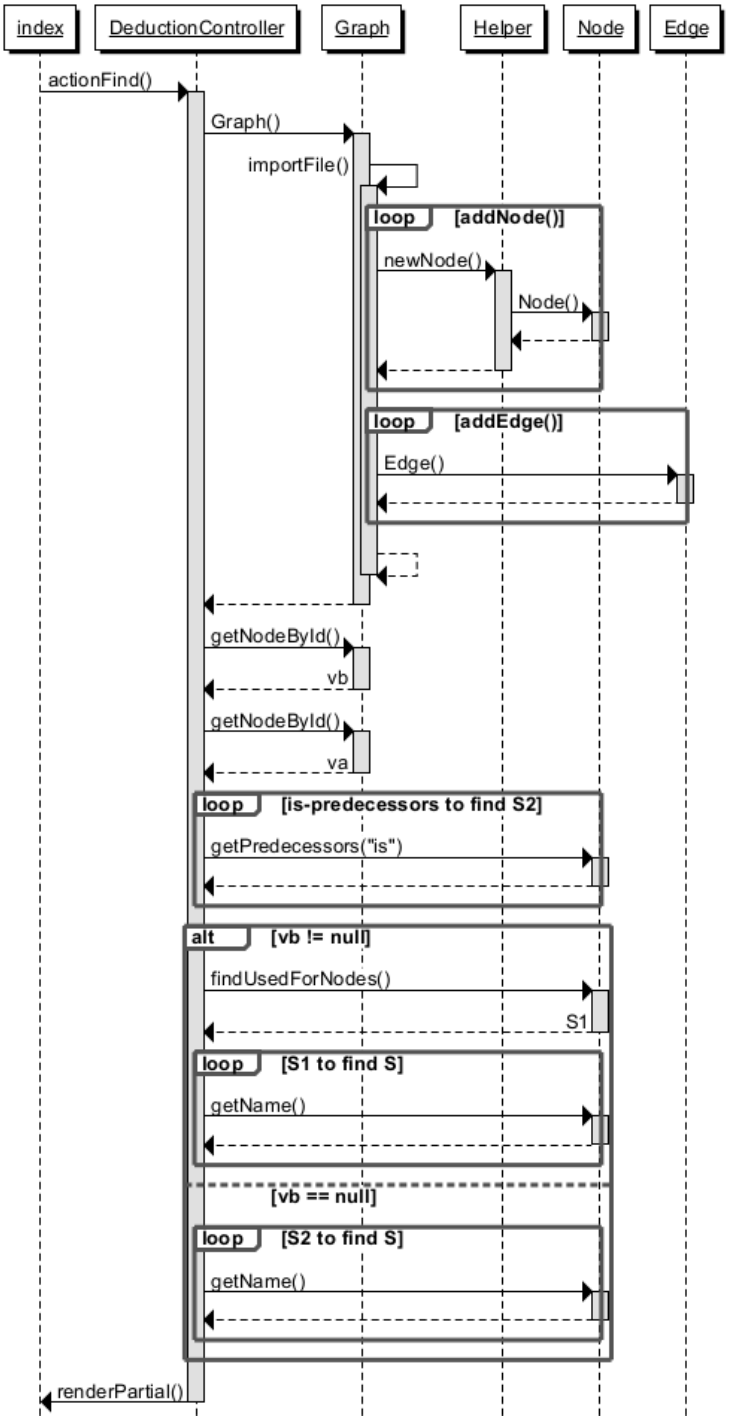


Abb. 39 Sequenzdiagramm zum Find!-Algorithmus

Die Sicht auf das Laufzeitverhalten des Algorithmus im Prototyp wird in Abb. 39 dargestellt. Im Vergleich zum Laufzeitverhalten des Isn't-It?-Algorithmus fällt hierbei die deutlich geringere Komplexität während der Deduktion auf. Auch bei diesem Algorithmus dient der Aufruf der Action *actionFind()* als Einstieg in die Implementierung. Wie im vorangegangenen Beispiel wird auch hier zunächst ein *Graph*-Objekt mit allen enthaltenen Knoten und Kanten erzeugt. Anschließend werden die Knoten *vb* und *va* durch Aufruf der

Methode *getNodeById()* initialisiert. Sie entsprechen der gesuchten Klasse (*va)* und der gesuchten Aufgabenstellung (*vb*).

Sind diese Elemente gefunden, erfolgt die Suche nach allen Abstraktionen von *va*. Dazu wird in einer Schleife die Funktion *getPredecessors()* unter Angabe des Parameters „is“ ausgeführt und die Ergebnismenge als *S2* gespeichert. Die anschließende Unterteilung in die Alternativen *vb != null* und *vb == null* ermöglicht einem Anwender, auf die Angabe einer konkreten Aufgabenstellung zu verzichten. In diesem Fall würde der Algorithmus schlicht alle Konkretisierungen von *va* ausgeben, indem die gesuchte Menge *S* alle Elemente aus *S2* übertragen bekommt. Wurde vom Nutzer eine Aufgabenstellung angegeben (*vb != null)*, wird zunächst nach allen „used-for“-Vorgängern von *vb* gesucht und in *S1* gespeichert. Die Suche erfolgt über die Methode *findUsedForNodes()*. Im nächsten Schritt wird die Schnittmenge aus *S1* und *S2* gebildet und als *S* gespeichert. Schließlich erfolgt die Ausgabe aller in *S* enthaltenen Elemente durch die Methode *renderPartial()*, welche die Menge an die *index*-View übergibt und somit auf dem Ausgabegerät des Anwenders darstellt.

5.1.3 Test

Der Prototyp wird in zwei Schritten auf Korrektheit getestet. Im ersten Schritt erfolgt eine Überprüfung der Benutzeroberfläche und damit der Funktionen, die vom Nutzer ausgelöst werden und eine Veränderung der grafischen Oberfläche nach sich ziehen. Die Testfälle sind in Tabelle 19 im Anhang ab Seite xxv tabellarisch aufgelistet. Sie umfassen insgesamt 26 Tests. Neben einer Beschreibung des durchzuführenden Verhaltens und den jeweiligen Verweisen zur zugehörigen Anforderung, werden für jeden Testfall Akzeptanzkriterien aufgezeigt, die zur Überprüfung einer erfolgreichen Testdurchführung dienen. Tabelle 11 zeigt den Aufbau der Testfälle am Beispiel des Testfalls mit der Id „T22“. Er dient zur Überprüfung der Anforderung „A07“, welche auf die Deduktionsmöglichkeiten des Prototyps Bezug nimmt. Die Testfälle bauen aufeinander auf, weshalb in der Beschreibung von T22 auf weitere Testfälle verwiesen wird. Die erwähnten Akzeptanzkriterien werden als „Erwartetes Ergebnis“ erfasst.

Im zweiten Schritt des Tests werden wesentliche Quellcode-Elemente auf ihr erwartetes Verhalten überprüft. Dabei erfolgt die Fokussierung auf Methoden und Klassen, die bei den beiden implementierten Algorithmen „Isn’t-It?“ und „Find!“ verwendet werden. Zur

Durchführung dieser Testfälle wird auf Testdaten zurückgegriffen, die in Abb. 51 bis Abb. 56 (Anhang ab Seite xxxii) in strukturierter Form aufgezeigt werden. Visualisiert nach den Vorgaben des Modells entsprechen diese Testdaten den in Abb. 40 gezeigten Elementen.

Tabelle 11 Beispiel eines Testfalls aus Tabelle 19 (Anhang, Seite xxv)

Id	A.-Id	Testfall
T22	A07	Nutzer lädt die von ihm erstellte Ontologie mit dem booleschen Ausdruck aus T20. Anschließend klickt er auf den Button "Isn't it?". Im Dialog wählt er die Cell aus T01 durch Eingabe des entsprechenden Namens und bestätigt den Dialog.
		Erwartetes Ergebnis: Ein Dialog öffnet sich mit einem Textfeld zur Eingabe eines Elementnamens. Nach Eingabe eines Buchstabens öffnet sich ein Autocomplete-Feld zur Auswahl eines Elementes. Nach Bestätigen des Dialogs öffnet sich ein neues Dialogfenster zur Eingabe des Feature-Wertes aus T03. Darin enthalten ist ein Textfeld und die Fragestellung der Question aus T06.

Insgesamt 12 Testfälle sollen die wesentlichen Funktionen prüfen, die zur Durchführung der Deduktionsalgorithmen notwendig sind. Sie werden in Tabelle 20 (Anhang, Seite xxix) aufgelistet.

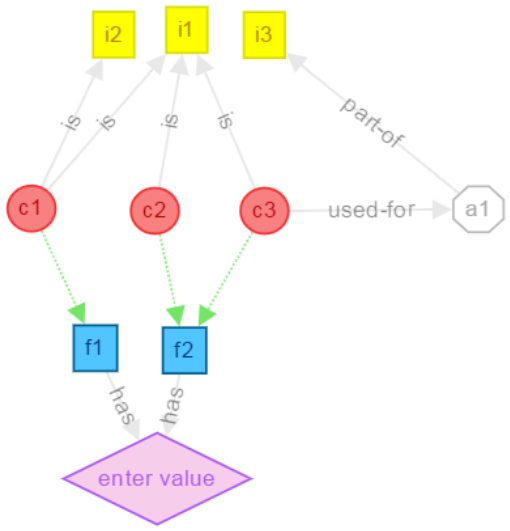


Abb. 40 Visualisierung der Testdaten zur Durchführung von Quellcode-Tests

Tabelle 12 enthält einen Auszug dieser Testfälle und zeigt an den Beispielen T34 und T35, wie die Beschreibung der Testfälle zu verwenden ist. Die Spalte „Klasse“ gibt das be-

troffene Sourcecode-Element an, welches die zu testende Methode enthält. Jeder Testfall beinhaltet neben dem auszuführenden Code auch ein erwartetes Ergebnis, das für einen erfolgreichen Test zutreffen muss. Testfall T34 testet beispielsweise die Funktion *findUsedForNodes()* der Klasse *Node*. Dafür ist zunächst die Initialisierung eines *Graph*-Objektes erforderlich, was mit den Testdaten aus der Datei „test_ontology_2.json“ befüllt wird. Anschließend wird der Knoten I1 gesucht und in die Variable $a gespeichert. Für einen erfolgreichen Test muss $b einem 1-dimensionales Array mit genau einem Element, der Cell C3, entsprechen. Das Setzen eines Ergebnisses und damit auch das Versenden an alle betroffenen Observer werden in T35 getestet. F1 wird ein Ergebnis zugewiesen. Da sich das Ergebnis von C1 aus dem Ergebnis von F1 zusammensetzt, kann es gebildet werden. Der boolesche Ausdruck zur Ergebnisbildung von C1 lautet „r(F1) > 5“, was bei T35 zu einem false-Wert führt. Da das Ergebnis von I2 vom C1-Ergebnis bestimmt wird, kann auch dieses Ergebnis gebildet werden. Da C1 false ist, ist auch I1 false. Alle anderen Ergebnisse lassen sich nicht ableiten, solange F2 noch kein Ergebnis besitzt (Testfall T36).

Tabelle 12 Auszug aus Tabelle 20 (Anhang, Seite xxix)

ID	Klasse	Testfall
T34	Node	$graph = new Graph("test_ontology_2.json"); $a = $graph->getNodeById("I1"); $b = $a->findUsedForNodes();
		Erwartetes Ergebnis: $b ist ein Array, das alle relevanten used-for-Knoten von I1 enthält: C3
T35	Node	$graph = new Graph("test_ontology_2.json"); $a = $graph->getNodeById("F1"); $a->setResult("5");
		Erwartetes Ergebnis: Das Ergebnis von $a ist 5. Das Ergebnis von C1 ist false. Das Ergebnis von I2 ist false. Alle anderen Knoten des Graphs besitzen noch kein Ergebnis (null).

Insgesamt wurden 38 Testfälle zur Überprüfung des Prototyps definiert. Die Testfälle T01 bis T24a wurden von einem Anwender geprüft. Alle Akzeptanzkriterien konnten erfolgreich getestet werden. Auch die Testfälle zur Überprüfung des Source-Codes T25 bis T36 konnten erfolgreich durchgeführt werden. Dabei entsprachen alle Ergebnisse den Erwartungen. Der Prototyp wurde demnach erfolgreich auf Korrektheit getestet.

5.2 Evaluierungsschritt I: Anwendbarkeit des Modells

Im ersten Schritt der Evaluierung wird die Anwendbarkeit als Modellierungsmethode getestet. Dazu modelliert ein unerfahrener Nutzer Wissen am Beispiel von Scrum mithilfe der vorgestellten Modellkomponenten. Wesentliches Ziel dabei ist, die Notwendigkeit und Vollständigkeit aller enthaltenen Element- und Assoziationsklassen zu überprüfen. Ein weiteres Ziel dieses Schrittes verfolgt die Herausstellung von Erweiterungspotenzial und Schwachstellen, die in Verbindung mit der Modellierung auftreten.

5.2.1 Vorgehensweise

Die Modellierungsmethode durchlief im Laufe ihrer Entstehung verschiedene Entwicklungs- und Erweiterungsschritte. Zur Überprüfung der aktuellen Methode, wird einem, bezüglich der Modellierungsmethode unerfahrenen, Anwender eine vorherige Entwicklungsstufe übergeben. Dem Probanden ist zum Zeitpunkt der Anwendung nicht bewusst, dass er eine ältere Version nutzt. Um die aktuelle Version der Modellierungsmethode zu untermauern, müsste der Proband mit der älteren Version auf Probleme stoßen, die durch die erweiterte Methode behoben sind.

Die Anwendbarkeit wird anhand einer konkreten Modellierungsaufgabe überprüft. Dazu wird fachspezifisches Wissen von einem fachfremden Anwender modelliert. Im Rahmen einer studentischen Arbeit sollen diesbezüglich Probleme und Fragen im Umgang mit der Modellierungsmethode aufgedeckt werden. Es erfolgt zunächst eine selbständige Einarbeitung in die Modellierungsmethode. Am Beispiel der agilen Softwareentwicklungsmethode Scrum wird anschließend vom Probanden eine Ontologie erstellt und Schwierigkeiten, die während der Modellierung aufgetreten sind, erfasst. Zur Erstellung der Ontologie werden drei Literaturquellen über Scrum sukzessiv bearbeitet und mit jedem Bearbeitungsschritt die Wissensbasis verfeinert und optimiert. Die Ergebnisse dieses Praxistests werden im Folgenden zusammengefasst.

5.2.2 Ergebnisse

Wie beschrieben, wird die aktuelle Version des Modells durch die Anwendung einer älteren Version gegenüber gestellt. Aufgetretene Probleme und Fragestellungen, die sich aus der Anwendung der älteren Variante ergeben haben, sind in Tabelle 13 aufgelistet. Die

Tabelle enthält neben der Problemstellung auch eine Problemlösung und eine Angabe, ob das Problem in der aktuellen, hier vorgestellten Modellversion behoben wurde.

Tabelle 13 Probleme während der Anwendung (nach ORTLEPP 2016)

Aufgedeckte Probleme	**Problemlösung**	
Assoziationsklassen sind ausschließlich durch Pfeilspitzen voneinander abgegrenzt (bspw. *is-*Assoziation ⊕). Dies führt zu schlechter Lesbarkeit.	Assoziationsklassen im Klartext auf die Kante schreiben und die Pfeilspitze vereinheitlichen	☑
Elementklassen werden ausschließlich durch geometrische Formen voneinander abgegrenzt (bspw. *Feature* □ und *Item* 🗋). Dies führt zu Verwechslungsgefahr.	Deutliche Abgrenzung anhand farblicher Kennzeichnung	☑
Nur Elemente benachbarter Ebenen konnten miteinander assoziiert werden (bspw. *Cell* und *Feature*, aber nicht *Item* und *Feature*). Dies führte zu Problemen der Wissensabbildung.	Eine Assoziation kann zwischen jeder Elementklasse gebildet werden.	☑
Dem Anwender war die Verwendung von Einzahl und Mehrzahl unklar (bspw. *Rolle* vs. *Rollen*)	Mengen können über Positive-Constraints zu Input-Feature abgebildet werden. Es obliegt jedoch dem Anwender ein Wissenselement mit einer Mehrzahl zu bezeichnen.	☑
Die Modellierung von Aktivitäten war nicht möglich (bspw. *Product Owner sichtet Eigenschaft*).	Integration von Activities und can-Assoziation	☑
Negationen konnten nicht abgebildet werden (bspw. *Scrum Master is not Product Owner*).	Abbildung von Negation kann über Negative-Constraints erfolgen oder durch die *-not-Assoziationsklasse.	☑
Die Modellierung zeitlicher Abfolgen ist nicht möglich (bspw. *Sprint Planung zu Beginn des Sprints*).	Eine Abbildung ist zwar mittels Constraints möglich, aber nicht die ideale Lösung.	☐
Nutzung von Synonymen ermöglichen. (*Umsetzungsteam* und *Entwicklerteam*)	Einführung der Assoziationsklasse *same-as*.	☑

Darstellung von gemeinsamen Aktivitäten zwischen verschiedenen Rollen war nicht möglich (bspw. *Product Owner can zusammenarbeiten mit Customer*).	Einführung zusammengesetzter Wissenselemente wie Activity und Objekt durch die Elementklasse *Combining*	☑
Bei Benennung einer Aktivität kann es zu Redundanzen kommen (bspw. *durchführen* und *ausführen*).	Die *same-as*-Assoziation verdeutlicht zwar diesen Sachverhalt. Der Nutzer erfasst ihn jedoch nicht immer händisch. Daher ist die Entwicklung einer Erkennung von Duplikaten erforderlich.	☐
Vermischung von deutschen und englischen Wörtern führt zu Redundanzen.	Den Grundstein bildet erneut die *same-as*-Assoziation. Es muss jedoch eine Integration von Sprachzentren ermöglicht werden.	☐
Regeln zu Transformationsschritten (Änderungen) von Wissen in einer bereits vorhanden Ontologie	Integration von STM und LTM – Techniken.	☑

Die aufgetretenen Probleme konnten nahezu vollständig mit der in dieser Arbeit vorgestellten Version des Modells gelöst werden. Erweiterungspotenzial bietet insbesondere eine Modellerweiterung zur Integration verschiedener Sprachen. Zudem kann derzeit nur bedingt eine Erfassung von zeitlichen Abfolgen in einer Ontologie erfolgen. Möglichkeiten, um diese zeitlichen Abhängigkeiten zu modellieren, werden vom Probanden als besonders wertvoll eingeschätzt.

Nach Vollendung des Praxistests wird die Komplexität der Modellierungsmethode vom Probanden als besondere Herausforderung erachtet: „[…] erst bei einer intensiven Auseinandersetzung mit der Modellierungsmethode [entsteht] das Verständnis für die Ebenen und Verbindungen [...]". Es wird weiterhin erwähnt, „dass der unerfahrene Nutzer zunächst zahlreiche Ausdrücke mit der Modellierungsmethode formulieren muss, um einen sicheren Umgang damit erlangen zu können und somit die Ausdrücke korrekt zu formulieren". (ORTLEPP 2016, S. 14)

5.3 Evaluierungsschritt II: Schlussfolgerung

Im zweiten Schritt der Evaluierung werden die Deduktionsalgorithmen und die aus ihnen hervorgehenden Ergebnisse validiert. Die dabei verwendete Methode ist eine Gegenüberstellung einer Expertenentscheidung mit einer modellbasierten Entscheidung, bei der spe-

zielle Tools zur Softwareentwicklung unter Berücksichtigung verschiedener Kriterien ausgewählt werden sollen.

5.3.1 Vorgehensweise

Wie in Abb. 41 zusammengefasst, besteht das Vorgehen aus insgesamt fünf Teilprozessen.

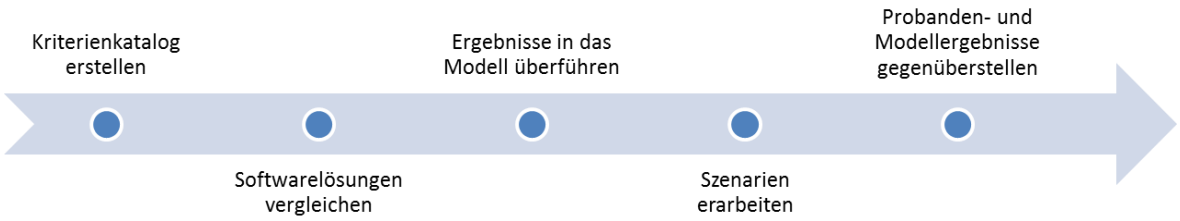


Abb. 41 Vorgehen bei Evaluierungsschritt II

Der Teilprozess „Kriterienkatalog erstellen“ hat zum Ziel, eine Methode zu entwickeln, um Softwareprodukte auf ihren Nutzen für Softwareentwicklungsprojekte anhand ausgewählter Kriterien prüfen zu können. Die Auswahl der Kriterien erfolgt anhand einer nach WEBSTER UND WATSON durchgeführten Literaturanalyse. In „Softwarelösungen vergleichen“ sollen auf Basis des im ersten Teilprozess erarbeiteten Kriterienkataloges vier ausgewählte Softwarelösungen in der Softwareentwicklung auf Vorhandensein und Ausprägung von Kriterien verglichen werden. Anschließend erfolgt die Übertragung des Kriterienkataloges und der Ergebnisse des Softwarevergleichs in den modellbasierten Prototypen. Dazu werden die notwendigen Wissensartefakte durch Erstellen von Element- und Assoziationsklassen formalisiert. Der nächste Teilschritt „Szenarien erarbeiten“ dient zur Schaffung einer einheitlichen Datenbasis zur späteren Durchführung der Experten- und Modellentscheidungen. Dabei werden unterschiedliche Anforderungen und Rahmenbedingungen entworfen, was als Ausgangssituation für den letzten Teilprozess „Probanden- und Modellergebnisse gegenüberstellen“ dient. Dieser soll schließlich die Richtigkeit der Modellergebnisse nachweisen, indem auf Basis der vorangegangenen Teilprozesse sowohl Probanden als auch das Modell, unabhängig voneinander, geeignete Software bestimmen. Die Ergebnisse werden in einem Fazit gegenüberstellt und analysiert.

5.3.2 Kriterienkatalog erstellen

Ziel des Kriterienkataloges ist die Analyse und Vergleichbarkeit von Softwarelösungen, die zur Softwareproduktentwicklung eingesetzt werden. Er enthält zahlreiche planungsre-

levante Kriterien auf technischer und organisatorischer Ebene, die funktional Anwendung in derzeitigen Softwarelösungen finden.

A	2			Anforderungsdokumentation
A	2	1		Erfassen der Akzeptanzkriterien
A	2	2		Funktionale Anforderungen
A	2	3		Nichtfunktionale Anforderungen
A	2	4		Erstellung von Wireframes
A	2	5		Templateunterstützung (z.B. Volere)
A	2	6		auto. Generierung Lasten-/Pflichtenheft
A	2	7		Erstellen von Use-Case-Diagrammen
A	3			Anforderungsverwaltung
A	3	1		Identifikation als Projektitem
A	3	2		Allgemeine Lebensstati
A	3	3		Strukturierung der Anforderungen
A	3	4		Anfordungsschätzung
A	3	5		Priorisierung der Anforderungen
A	3	6		Kategorisierung der Anforderungen
A	3	7		Ticketsystem
A	3	8		Änderungshistorie
A	3	9		Filtern von Anforderungen
E				**Projektmanagement**
E	1			Benutzerverwaltung
E	1	1		Benutzerprofile
E	1	2		Benutzerrechte

Abb. 42 Ausschnitt aus Kriterienkatalog (TRIEBEL 2015)

Zur Erarbeitung des Kriterienkataloges wurde im Rahmen einer studentischen Arbeit von TRIEBEL eine Literaturanalyse nach WEBSTER UND WATSON durchgeführt. Auf dieser Basis erfolgte die Auswahl der Kriterien, die nach den Phasen des Softwareentwicklungsprozesses kategorisiert wurden. Das Ergebnis enthält neben technischen und organisatorischen Faktoren, auch koordinatorische und kostenbezogene Rahmenbedingungen, in denen sich Softwarelösungen differenzieren.

In Form von Kriterienbäumen wurden die unterschiedlichen Kriterien und ihre Ausprägungen ermittelt und visualisiert. Anschließend erfolgte die Überführung in tabellarischer Form, um die Vergleichbarkeit von Softwarelösungen besser gewährleisten zu können. Die Kriterien wurden in sieben Hauptkategorien und insgesamt 30 Unterkategorien unterteilt. Im Rahmen des Evaluierungsschritt II der vorliegenden Arbeit werden vier aus sieben Hauptkategorien ausgewählt: Anforderungsmanagement, Projektmanagement, Systemvoraussetzungen und Kosten. Einen Ausschnitt aus dem tabellarischen Kriterienkatalog enthält Abb. 42. Alle für diese Arbeit relevanten Kriterien sind in Tabelle 18 (Anhang, Seite xxi) enthalten.

5.3.3 Softwarelösungen vergleichen

Anhand der im vorherigen Abschnitt erarbeiteten Kriterien werden in diesem Abschnitt die Ergebnisse eines Softwarevergleichs präsentiert. Für den Vergleich wurden vier Softwarewerkzeuge ausgewählt. Atlassian Jira als Projektplanungs- und Projektverfolgungs-Tool wird in allen relevanten Kategorien (Projektmanagement, Systemvoraussetzungen, Kosten und Anforderungsmanagement) untersucht. Eclipse ist eine integrierte Entwicklungsumgebung (IDE) und wird nachstehend auf die Attribute der Kategorien Projektmanagement, Systemvoraussetzungen und Kosten analysiert. Gleiches erfolgt mit den Softwaretools XStudio, einer Test Management Software, sowie Microsoft Visio als Softwaremodellierungsanwendung. Die Resultate der Analyse befinden sich vollumfänglich in Tabelle 18 (Anhang, Seite xxi), einen Ausschnitt bietet Abb. 43.

E	2	2			Chat	HipChat	-	-	-
E	2	3			Kommentare	x	-	-	-
E	2	4			Anhängen von Dateien	x	-	x	x
E	2	5			Reminder	x	-	-	-
E	2	6			Push-Nachrichten (eMail)	x	-	-	-
E	2	7	1		Export/Import CSV	x	-	x	-
E	2	7	2		Export/Import Word	x	-	-	x
E	2	7	3		Export/Import PDF	x	-	-	x
E	3				Projektstrukturplan	100,00%	20,00%	0,00%	0,00%
E	3	1			Ziele	x	-	-	-
E	3	2			Meilensteinplanung	x	-	-	-
E	3	3			Zeiten/Dauer	x	Rabbit	-	-
E	3	4			Ressourcenmanagement	x	-	-	-
E	3	5			Kosten-/Budgetmanagement	x	-	-	-
E	4				Projektkontrolle	x	x	x	-
E	5				Softwareentwicklungsmethode	50,00%	100,00%	100,00%	100,00%
E	5	1			Wasserfall	-	x	x	x
E	5	2			Scrum	x	x	x	x
E	6				Schnittstellenverfügbarkeiten	100,00%	100,00%	50,00%	50,00%
E	6	1			Schnittstelle zu ERP Systemen	x	x	?	x
E	6	2			Schnittstelle zu verwendetes Ticketsyste	x	x	x	?
X					**Systemvoraussetzungen**				
X	1				Arbeitsspeicher/RAM	2 GB	1 GB	1 GB	1 GB
X	2				Festplattenspeicherplatz	1 GB	1 GB	?	2 GB
X	3	1	1		CPU Intel	x	x	?	?

Abb. 43 Ausschnitt aus dem Softwarevergleich anhand des Kriterienkataloges

Für die im späteren Verlauf dieses Evaluierungsschrittes erfolgende Erarbeitung der Fallstudie wurde die Software Jira als einzige in der Kategorie „Anforderungsmanagement“ auf ihre Funktionalitäten untersucht. Hingegen erfolgte eine tatsächliche Gegenüberstel-

lung aller vier Softwareprodukte in den Kategorien Projektmanagement, Systemvoraussetzungen und Kosten. Die Fallstudie soll zwei Anwendungsfälle abdecken. Zum einen soll eine vorgegebene Softwarelösung auf ihre Eignung als Hilfsmittel zur Durchführung eines beliebigen Schrittes innerhalb des Softwareentwicklungsprozesses geprüft werden. Dafür erfolgte die Analyse der Kriterien Anforderungsmanagement bezüglich der Software Jira. Der zweite Anwendungsfall basiert auf dem Direktvergleich der Softwarelösungen in den Kategorien Projektmanagement, Systemvoraussetzungen und Kosten. Hier sollen Probanden anhand verschiedener Vorgaben die passende Softwarelösung herausfinden können.

5.3.4 Daten in das Modell überführen

In diesem Teilschritt werden die Erkenntnisse aus den beiden vorangegangenen Abschnitten in das in dieser Arbeit vorgestellte Modell überführt. Dazu werden zunächst die erarbeiteten Kriterien in Form von Items, Cells, Feature und Questions über den in Abschnitt 5.1 (Seite 89) gezeigten Prototypen angelegt. Anschließend erfolgt die Überführung des Softwarevergleichs, bei der die einzelnen Softwarewerkzeuge in die Beispielontologie integriert werden. Diesen Elementen werden daraufhin Positive- und Negative-Constraints zur Ergebnisbildung hinzugefügt.

Abb. 44 zeigt auszugsweise, wie der Softwarevergleich und die Kriterien in Form des Modells visualisiert werden konnten. Einen Eindruck vom gesamten Umfang dieser Überführung bietet Abb. 48 (Anhang, Seite xxiii). Die Kategorie „Anforderungsmanagement“ und ihre zugehörigen Unterkategorien wurden als Items angelegt. Einzelne Kriterien dieser Kategorie konnten meist als Combining aus einer Activity und einem Item erfasst werden, was in der oberen Hälfte von Abb. 44 deutlich wird. Die Combinings wurden dann als part-of-Vorgänger der jeweils zugehörigen Unterkategorie zugewiesen. Auf diese Weise konnte die hierarchische Struktur der Kriterienbäume erreicht werden.

Die einzelnen Softwaretools wurden dann durch used-for-Assoziationen mit diesen Combinings verknüpft. Somit erfolgte eine eindeutige Zuordnung eines konkreten Softwarewerkzeugs mit einem funktionsbezogenen Kriterium. Beispielsweise ergeben sich aus den Informationen

a1. *‚kategorisieren anforderungen‘ part-of anforderungsverwaltung*
b1. *anforderungsverwaltung part-of anforderungsmanagement*

c1. *jira used-for ‚kategorisieren anforderungen‘*

folgende Wissenszusammenhänge:

a2. Teilgebiet des Anforderungsmanagement ist Anforderungsverwaltung.

b2. Zur Anforderungsverwaltung zählt das Kategorisieren von Anforderungen.

c2. Jira kann zum Kategorisieren von Anforderungen genutzt werden.

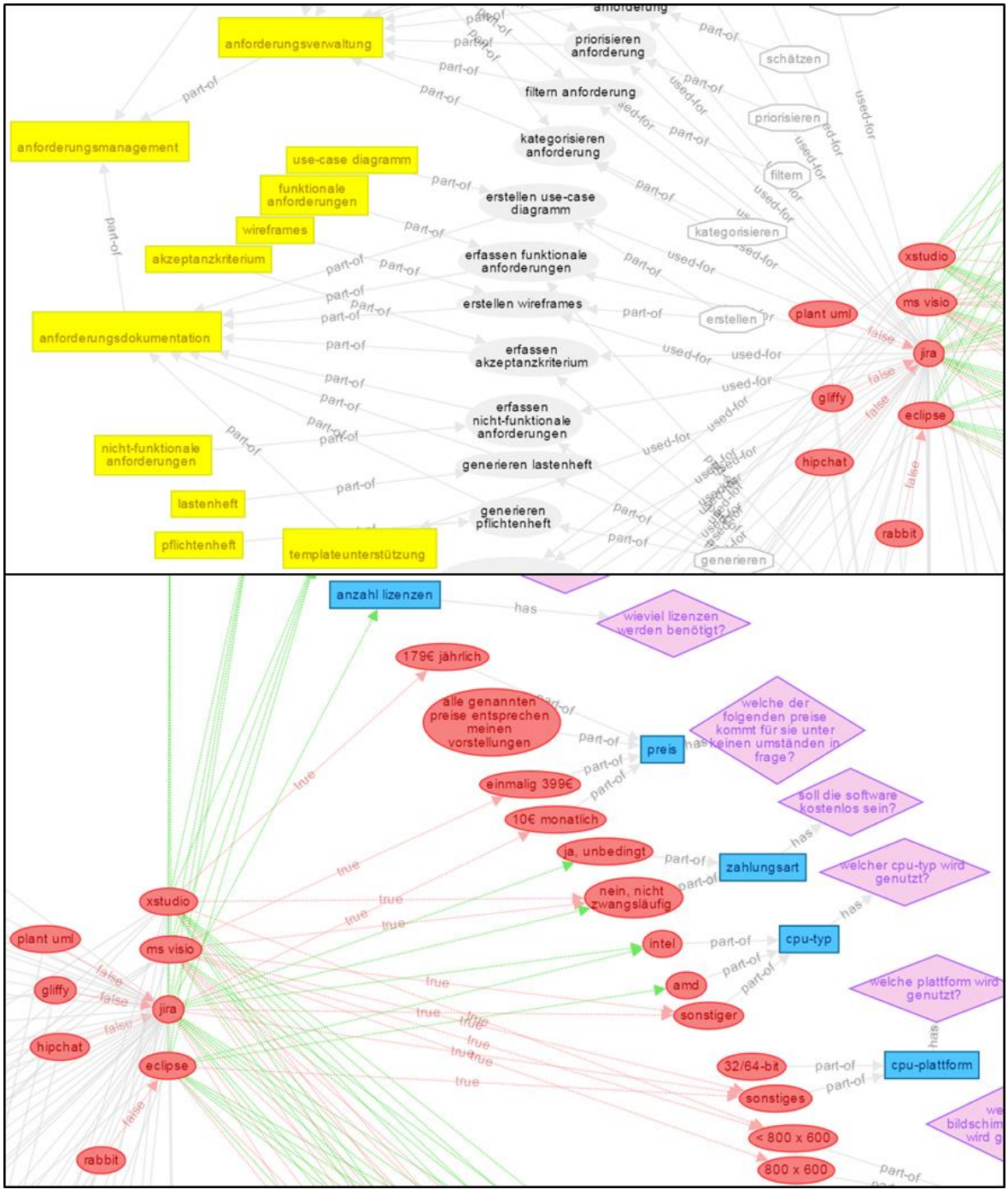


Abb. 44 Ausschnitt aus dem überführten Softwarevergleich

Kriterien, die keine funktionalen Anforderungen an die Software stellen, benötigten eine andere Form der Modellierung. Hier wird nicht der Nutzen (used-for) der einzelnen Softwaretools modelliert, sondern es erfolgt eine Modellierung von unterschiedlichen Abhängigkeiten zu konkreten Ausprägungen (beispielsweise Systemvoraussetzungen). Diese Zusammenhänge werden, wie im unteren Abschnitt von Abb. 45 zu sehen, durch Positive- und Negative-Constraints erzeugt. Die Kriterien als solche, wie CPU-Typ oder Preis, wurden während der Überführung als Feature erfasst. Diese erfolgte in Hinblick auf die später stattfindende automatisierte Schlussfolgerung zur anwendungsfallabhängigen Auswahl der Softwaretools. Durch Verknüpfung mit Questions können die jeweiligen Systemvoraussetzungen oder Budgetvorstellungen seitens der Anwender erfragt werden und somit in die Deduktionsalgorithmen einfließen.

5.3.5 Szenarien erarbeiten

Ziel dieser Evaluierung ist, zu beweisen, dass trotz zwei verschiedener Vorgehensweisen (empirische Befragung und automatische Deduktion durch das erarbeitete Modell) dieselbe Lösung gefunden wird. Um diesen Beweis durchzuführen, werden einheitliche Szenarien definiert, die beiden Herangehensweisen als Basis dienen. Tabelle 14 enthält zehn Szenarien und die jeweiligen erwarteten Ergebnisse. Die Szenarien 1-5 sollen den „Find!“-Abstraktionsalgorithmus (Abschnitt 4.1.2, Seite 76) evaluieren. Die Szenarien 6-10 prüfen den Konkretisierungsalgorithmus „Isn‘t-It?“ (Abschnitt 4.1.1, Seite 73).

Tabelle 14 Szenarien mit erwartetem Ergebnis

Szenario		**Erwartetes Ergebnis**
1	Welche Software dient zur Anforderungsverwaltung?	Jira
2	Welche Software dient zur Erstellung von Wireframes?	Gliffy
3	Mit welcher Software lassen sich funktionale Anforderungen erfassen?	Jira
4	Welche Software bietet eine Benutzerverwaltung?	Jira, XStudio
5	Welche Software deckt alle Teilgebiete des Anforderungsmanagement ab?	keine
6	Prüfe ob Jira unter den Bedingungen aus Tabelle 15 (A) geeignet ist.	geeignet
7	Prüfe ob Jira unter den Bedingungen aus Tabelle 15 (B) geeignet ist.	ungeeignet

8	Prüfe ob Eclipse unter den Bedingungen aus Tabelle 15 (A) geeignet ist.	ungeeignet
9	Prüfe ob Eclipse unter den Bedingungen aus Tabelle 15 (B) geeignet ist.	geeignet
10	Prüfe ob MS Visio unter den Bedingungen aus Tabelle 15 (B) geeignet ist.	geeignet

Um die Szenarien zur Eignungsprüfung (Szenarien 6-10) nutzen zu können, müssen Rahmenbedingungen definiert werden, die dem Ist-Zustand eines Umweltausschnittes entsprechen. Tabelle 15 beinhaltet neun Rahmenbedingungen, die sich aus den Kriterien für die Software-Systemvoraussetzungen ergeben. Es werden zwei fiktive Zustände (A und B) aufgelistet, welche sich in den Rahmenbedingungen 1, 2 und 8 unterscheiden.

Tabelle 15 Rahmenbedingungen für Szenarien S5-S8

Rahmenbedingung		**A**	**B**
1	Verwendetes Betriebssystem	MAX OS X 10.9	Windows 7
2	Verwendeter Arbeitsspeicher	4 GB	1 GB
3	Verwendeter Festplattenspeicher	1 TB	1 TB
4	Verwendeter CPU Chipsatz	Intel	Intel
5	Verwendete CPU Plattform	32/64-bit	32/64-bit
6	Verwendete CPU Leistung	4 Ghz	4 Ghz
7	Verwendete Bildschirmauflösung	1920x1080 px	1920x1080 px
8	Gewünschte Laufzeitumgebung	webbasiert	Lokal
9	Preis	beliebig	beliebig

5.3.6 Ergebnisse gegenüberstellen

Um die Qualität der Deduktionsalgorithmen zu testen, wurden die Szenarien zeitgleich von 15 Probanden sowie durch den Prototyp auf Basis der gegebenen Rahmenbedingungen und des Kriterienkataloges beantwortet. Ziel dabei war der Beweis, dass die implementierten Deduktionsalgorithmen zu Entscheidungen führen, die dem Menschen gleichen.

Für die empirische Befragung wurden zufällig 15 Studierende während einer Vorlesung ausgewählt. Ihre Aufgabe bestand darin, eine völlig selbstständige Beantwortung der Szenarien und die Dokumentation der Lösungen vorzunehmen. Als Hilfsmittel diente ihnen

der erarbeitete Kriterienkatalog. Die Ergebnisse sind in Tabelle 16 in ihrer absoluten Häufigkeit aufgelistet.

Tabelle 16 Absolute Häufigkeit aller Antworten ohne Prototyp

Szenario	„Jira“	„Gliffy“	„XStudio“	„Visio“	„Ja“	„Nein“	„Keine“	Sonst.
1	15							
2	7	9						
3	15							
4	13		14	1				
5	7						8	
6					14	1		
7					1	14		
8					5	9		1
9					14			1
10					10	3		2

Im Kopf der Tabelle stehen alle Ausprägungen der von den Studierenden gegebenen Antworten. Dabei wurden die Antworten „Ja“ und „geeignet“, sowie die Antworten „Nein“ und „ungeeignet“ zusammengefasst. Unter Sonstiges fallen Antworten, die eine Mehrfachinterpretation zulassen (beispielsweise „?“). Die blau-markierten Felder entsprechen den erwarteten Antworten.

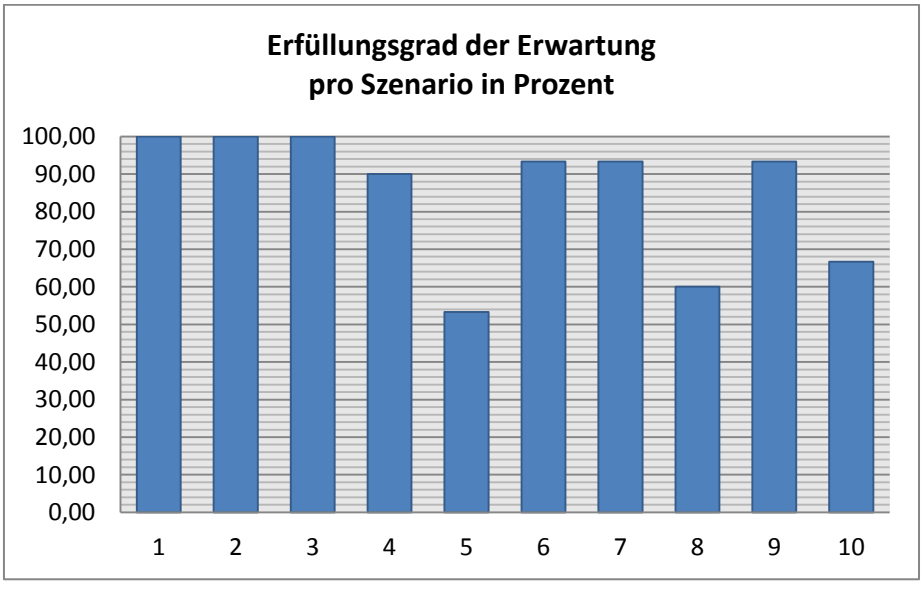


Abb. 45 Erfüllungsgrad der Erwartung

Bei Betrachtung der absoluten Häufigkeit und des Erfüllungsgrades aus Abb. 45 wird deutlich, dass sieben Szenarien (1, 2, 3, 4, 6, 7 und 9) die erwarteten Ergebnisse zu mindestens 90 Prozent erfüllen. Szenario 2 bildet in diesem Zusammenhang eine Besonderheit. Hier

kann die Antwort „Jira“ als zulässiges Ergebnis gewertet werden, da „Gliffy“ ein Add-On der Software darstellt. In Bezug auf die restlichen acht Szenarien können Abweichungen bis zu rund 48 Prozent (Szenario 5) von der Erwartung festgestellt werden. Hier konnten die Befragten also keine eindeutigen Lösungen finden, was einerseits an der Formulierung der Fragestellung und anderseits am subjektiven Wahrnehmungsverhalten der Befragten selbst liegen könnte.

Tabelle 17 Absolute Häufigkeit aller Antworten mit Prototyp

Szenario	„Jira“	„Gliffy“	„XStudio“	„Visio“	„Ja“	„Nein“	„Keine“
1	15						
2		15					
3	15						
4	15		15				
5							15
6					15		
7						15	
8						15	
9					15		
10					15		

Auch der Prototyp wurde von 15 Personen unbeeinflusst verwendet, um die Szenarien beantworten zu können. Im Unterschied zur vorherigen Befragung führten die Probanden die Lösungssuche ohne Kenntnis des Kriterienkataloges aus. Die Ergebnisse der Prototyp-Anwendung sind in Tabelle 17 dargestellt. Darin wird deutlich, dass keinerlei Abweichung zu den erwarteten Antworten zustande kam. Bei allen Szenarien konnten zu 100 Prozent die Erwartungshaltungen erfüllen werden.

Die dargebotenen Ergebnisse zeigen, dass die Verwendung des Prototyps zu einer dem Menschen ähnlichen Entscheidung führen kann und darüber hinaus Abweichungen in der Lösungssuche durch Vermeidung von subjektiven Wahrnehmungsverhalten verhindert.

6 Schlussbemerkungen

6.1 Zusammenfassung

Mit dieser Dissertation wurde ein Modell geschaffen, das Wissen aus der Domäne des Software Engineering repräsentiert und verarbeitet. Besonderes Ziel liegt dabei auf der Integration unterschiedlicher Störgrößen, welche die Umsetzung eines Softwareprojektes beeinflussen und den Projekterfolg gefährden. Solche Einflussfaktoren können sich beispielsweise aus projekt- und produktabhängigem, personen- und gruppenbezogenem oder aus ökonomisch bedingtem Hintergrund ergeben. Sie können in den verschiedenen Phasen eines Softwareprojektes auftreten und sich unterschiedlich, beispielsweise auf die Planung und Durchführung der Softwareentwicklung, auswirken. Neben der Integration von Einflussfaktoren, stellen die Visualisierung und Strukturierung von Wissen, sowie verschiedene Schlussfolgerungsmethoden weitere wesentliche Ziele des Modells dar.

Die formulierten Kernanforderungen an das zu erarbeitende Modell wurden zunächst mithilfe einer Literaturanalyse auf bereits existente Forschungsansätze geprüft. Die Literaturrecherche und die Analyse der erforderlichen Grundalgen zeigen, zur Umsetzung der definierten Ziele war die Erarbeitung eines neuen Modells erforderlich. Bestehende Forschungsarbeiten, deren Ansätze zur Erreichung der Zielstellung dieser Arbeit beitragen, sind in das entstandene Modell integriert. Auf diese Weise konnten die sechs Kernanforderungen wie folgt umgesetzt werden:

1. *Das im Modell enthaltene Wissen soll grafisch und mathematisch darstellbar sein, um einen hohen Grad an Formalisierung zu erreichen.*

Im Modell werden dem Anwender sechs verschiedene Elementklassen zur Verfügung gestellt, um Wissen zu modellieren. Durch grafische Darstellungsunterschiede können abstrakte und konkrete Wissenskonstrukte im Modell visualisiert werden. Zudem erfolgt eine Unterscheidung in Dateneingabe- und Datenüberführungsebenen. Zusätzlich erhält ein Modellanwender die Möglichkeit, mithilfe unterschiedlicher Assoziationsklassen Zusam-

menhänge zwischen Wissenselementen zu erzeugen. Eine mathematische Betrachtung im Sinne der Graphen- und Mengentheorie dient zur einheitlichen Beschreibung des Modells und trägt als wesentliches Merkmal zu Formalisierung bei.

2. *Es existieren Vorgaben zu verschiedenen Assoziationstypen, die zur Verknüpfung von Wissenselementen dienen. Eigene Assoziationstypen des Modellierers sind nicht möglich, um den Formalisierungsgrad zu stabilisieren.*

Der Modellanwender kann auf fünf Basisassoziationsklassen und fünf erweiterte Assoziationsklassen zurückgreifen, die in ihrer Gesamtheit in weiterführenden Arbeiten ergänzt werden können. Mithilfe dieser Assoziationsklassen lassen sich Anhängigkeiten wie Teil-Ganzes-Beziehungen, Objekt- und Subjekteigenschaften, Taxonomien, Fähigkeitszuweisungen, Gleichheitsbeziehungen, zeitliche Zustände oder moralische und ethische Zusammenhänge in das Modell integrieren. Das Erzeugen eigener Abhängigkeitsklassen ist nicht möglich. Der Anwender kann jedoch durch Anlegen sogenannter *Activities* beliebige Aktivitäten in Form von Wissensknoten im Modell verarbeiten. Im Gegensatz zu bisherigen Ontologiemodellen ist es auf diese Weise möglich, Aktivitäten nicht als Inferenzpfade zu betrachten, sondern sie zu Gegenständen der Inferenz selbst werden zu lassen.

3. *Ein weiteres Formalisierungsmerkmal sind vorgegebene Elementklassen, die dem Modellierer beim Überführen des Wissens zur Verfügung stehen, wie die Unterteilung in Konkretisierung und Abstraktion oder die Integration von Handlungen, Aufgaben oder Tätigkeiten.*

Ein Ziel des Modells ist die Integration der Knowledge Engineering Teilaufgaben. Diese umfassen das Akquirieren, Strukturieren und Visualisieren von Wissen. Das Akquirieren erfolgt im Wesentlichen auf zwei verschiedenen Wegen: (1) dem Erfassen von Daten und Überführen zu Informationen auf der *Data Source*- bzw. *Feature*-Ebene und (2) dem manuellen Erzeugen von Wissenselementen durch einen Modellierer. Diesem stehen neben den Elementklassen *Data Source* und *Feature* zwei weitere Kernklassen zur Verfügung: *Cells* und *Items*. Als *Cells* werden Konkretisierungen bzw. Instanzen abgelegt. Es handelt sich dabei um Wissenselemente, die keine weiteren Instanziierungen besitzen können. Eine Erweiterung der Elementklasse *Cell* bieten die sogenannten *Activities*. Sie dienen der Repräsentation von Fähigkeiten, Tätigkeiten, Aktivitäten, Aufgaben oder Handlungen. *Items*

sind Beschreibungen bzw. Abstraktionen, die mit Klassen im Sinne der Objektorientierung verglichen werden können.

4. *Es wird eine Möglichkeit dargeboten, hierarchisch vorliegende Wissenskonstrukte zu zusammenhängendem Wissen zu kombinieren.*

Aus der Kombination bereits existierender Wissenskonstrukte, beispielsweise aus der Zusammenführung einer *Cell* und einer *Activity*, kann Wissen extrahiert werden, welches ohne eine Zusammenführung nur implizit vorliegt. Diese Kombination aus mindestens zwei Wissenselementen wird im Modell als *Combining* bezeichnet. Jedes Combining kann wiederrum selbst als einzelnes Wissenselement in eine Inferenz integriert werden. Auf diese Weise können implizit vorliegende Wissensstrukturen zu explizitem Wissen überführt werden.

5. *Das Modell bietet die Möglichkeit, automatisiert aus dem modellierten Wissen Ableitungen, Entscheidungen und andere Schlussfolgerungen zu treffen, zum Beispiel durch Induktions- und Deduktionsalgorithmen.*

Um auf Basis des Modells vorhandenes Expertenwissen zur automatisierten Weiterverarbeitung nutzbar zu machen, werden fünf verschiedene Deduktionsalgorithmen zur Ableitung automatisierter Schlussfolgerungen vorgestellt. Diese können zur Lösung einer Software Engineering Fragestellung beitragen und somit den Entwicklungsprozess von Softwarelösungen beschleunigen oder Entscheidungsfindungen unterstützen. Die beiden komplexesten Algorithmen werden mithilfe eines Prototyps und auf Grundlage einer Probandenbefragung auf ihre Anwendbarkeit und Richtigkeit getestet. Darüber hinaus werden unterschiedliche Möglichkeiten dargestellt, um Wissen induktiv, automatisiert aus bereits modelliertem Wissen ableiten zu können.

6. *Das Modell liefert eine Möglichkeit, Datenerfassung bzw. Nutzereingaben zu modellieren und in Schlussfolgerungsprozesse zu integrieren.*

Zur Datenerfassung und Datenüberführung werden die zwei Ebenen *Data Source* und *Feature* genutzt. *Data Source* Elemente können Schnittstellen zu Sensoren enthalten, um beispielsweise Messwerte in einen modellbasierten Entscheidungsprozess einfließen zu lassen. Außerdem können Nutzerinteraktionen durch Integration von Multiple-Choice, Sin-

gle-Choice- und Input-*Feature* mit entsprechenden Fragestellungen (*Data Sources*) in Inferenzvorgänge einbezogen werden.

6.2 Ausblick und kritische Würdigung

Dem Modell kann aufgrund seiner regelbasierten Charakteristik bei automatisiert ablaufender Inferenz, des hohen Formalisierungsgrades und der generischen Architektur eine hohe Anwendungsbreite zugeschrieben werden. So sind Anwendungen auch außerhalb des Software Engineerings denkbar, beispielsweise in Domänen wie Medizin, Industrie oder Steuerungstechnik. Daraus ergeben sich weiterführende Untersuchungsfelder, aus denen zum Beispiel Parallelen zu Fuzzylogik und –reglern zur Integration von Hardwarekomponenten resultieren.

Hinsichtlich des in den letzten Jahren stark gewachsenen Interesses an der Verarbeitung großer Datenmengen (Big Data) können in weiterführenden Arbeiten Metriken zur Überprüfung von Skalierbarkeit eine wesentliche Rolle zur Potentialentfaltung des Modells beitragen. Diese Thematik betrifft neben der Untersuchung umfangreicher Ontologien, bzw. Ontologien mit hoher Knoten- und Kantenanzahl, auch die Verarbeitung von Echtzeitsensordaten mit hohen Datenraten. In diesen Fällen ist eine Analyse der vorgestellten Schlussfolgerungsalgorithmen bezüglich Verarbeitungszeit und Anzahl an Deduktionsschritten ein entscheidender Erfolgsfaktor bei der Verfeinerung der im Modell enthaltenen Inferenzmechanismen.

Neben der symbolischen KI des Modells zur regelbasierten Schlussfolgerung birgt insbesondere bei der Verarbeitung verschiedener Sensordaten, wie Bild- und Tondaten, eine weiterführende Untersuchung zur Integration neuronaler Strukturen Erweiterungspotenzial. Einen ersten Anknüpfungspunkt liefert der in dieser Arbeit vorgestellte Ansatz zur Verknüpfung des Modells mit künstlichen neuronalen Netzen.

Trotz einer durch das Modell geschaffenen Umsetzung aller definierten Kernanforderungen, besteht Potenzial bei der Implementierung, da der zu Evaluierungszwecken eingesetzte Prototyp die fünfte Kernanforderung nur teilweise unterstützt. Besondere Tragkraft hat daher eine im Rahmen zukünftiger Arbeiten durchgeführte Entwicklung einer Softwareumgebung, die die Anwendung aller im Modell existierenden Element- und Assoziationsklassen, sowie die Umsetzung aller erarbeiteten Deduktionsalgorithmen und Induktions-

mechanismen beinhaltet. Neben der Anwendung in industrieller Umgebung, kann eine derartige Softwarelösung auch zur Überprüfung der tatsächlichen Schlussfolgerungsgenauigkeit herangezogen werden. Dazu kann Expertenwissen über einen mittel- bis langfristigen Zeitraum modelliert werden und als Basis zur Validierung von Schlussfolgerungsergebnissen dienen, beispielsweise mithilfe eines Turing-Tests oder durch eine qualitative Expertenbefragung. Auf diese Weise kann der Evaluierungsansatz, wie er in dieser Arbeit verfolgt wird, erweitert werden.

Literaturverzeichnis

AHMAD, MOHAMMAD NAZIR (Hg.) (2013): Ontology-based applications for enterprise systems and knowledge management. Hershey, Pa.: Information Science Reference (Premier reference source).

ANAND, TARUN; GUPTA, PHALGUNI (1998): A Selection Algorithm for X + Y on Mesh. In: *Parallel Process. Lett.* 08 (03), S. 363–370. DOI: 10.1142/S0129626498000377.

ARP, ROBERT; SMITH, BARRY; SPEAR, ANDREW D. (2015): Building ontologies with basic formal ontology. Cambridge, Massachusetts: The MIT Press. Online verfügbar unter http://ieeexplore.ieee.org/servlet/opac?bknumber=7275982.

BECKER, BARBARA (2011): Leichter Lernen. Ein Erste-Hilfe-Paket für gestresste Eltern. Norderstedt: Books on Demand.

BEYDOUN, GHASSAN; LOW, GRAHAM; GARCÍA-SÁNCHEZ, FRANCISCO; VALENCIA-GARCÍA, RAFAEL; MARTÍNEZ-BÉJAR, RODRIGO (2014): Identification of ontologies to support information systems development. In: *Information Systems* 46, S. 45–60. DOI: 10.1016/j.is.2014.05.002.

BISSON, GILLES; NÉDELLEC, CLAIRE; CANAMERO, DOLORES (2000): Designing Clustering Methods for Ontology Building-The Mo'K Workbench. In: ECAI workshop on ontology learning, Bd. 31. Citeseer.

BISWAL, PURNA C. (2015): Discrete mathematics and graph theory. 4. Aufl. Delhi: PHI Learning Pvt. Ltd. Online verfügbar unter https://books.google.de/books?id=y2bkBQAAQBAJ.

BONABEAU, ERIC; THERAULAZ, GUY; DORIGO, MARCO (1999): Swarm intelligence. From natural to artificial isystems. New York: Oxford University Press. Online verfügbar unter

http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=144023.

BREITMAN, KARIN KOOGAN; CASANOVA, MARCO ANTONIO; TRUSZKOWSKI, WALTER (2007): Semantic Web: Concepts, Technologies and Applications. London: Springer-Verlag London Limited (NASA Monographs in Systems and Software Engineering). Online verfügbar unter http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10230263.

BREWSTER, CHRISTOPHER; WILKS, YORICK (2004): Ontologies, taxonomies, thesauri: Learning from texts. In: Proceedings of the Use of Computational Linguistics in the Extraction of Keyword Information from Digital Library Content Workshop.

BROOKS, RODNEY A. (1986): Achieving Artificial Intelligence through Building Robots. Hg. v. Massachusetts Institute Of Technology (AI-M-899).

BUITELAAR, PAUL; CIMIANO, PHILIPP; MAGNINI, BERNARDO (2005): Ontology Learning from Text. An Overview. In: Paul Buitelaar, Philipp Cimiano und Bernardo Magnini (Hg.): Ontology Learning from Text. Method, Evaluation and Applications. Amsterdam: IOS Press (Frontiers in artificial intelligence and applications, 123), S. 3–12.

CARDOSO, JORGE; SHETH, AMIT (2005): Introduction to Semantic Web Services and Web Process Composition. In: David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell et al. (Hg.): Semantic Web Services and Web Process Composition, Bd. 3387. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture notes in computer science), S. 1–13.

CHAN, C. W. (2002): Cognitive informatics: a knowledge engineering perspective. In: First IEEE International Conference on Cognitive Informatics. Calgary, Alta., Canada, 19-20 Aug. 2002, S. 49–56.

CHAN, C. W.; PENG, YAO; CHEN, LIN-LI (2002): Knowledge acquisition and ontology modelling for construction of a control and monitoring expert system. In: *International Journal of Systems Science* 33 (6), S. 485–503. DOI: 10.1080/00207720210133679.

CIMIANO, PHILIPP (2006): Ontology Learning and Population from Text. Algorithms, Evaluation and Applications. Boston, MA: Springer Science+Business Media LLC. Online verfügbar unter http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10155963.

CIMIANO, PHILIPP; VÖLKER, JOHANNA (2005): Text2Onto. In: David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell et al. (Hg.): Natural Language Processing and Information Systems, Bd. 3513. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture notes in computer science), S. 227–238.

COLLARD, MARTINI (Hg.) (2007): Ontologies-based databases and information systems. First and second VLDB workshops, ODBIS 2005/2006 ; Trondheim, Norway, September 2 - 3, 2005, Seoul, Korea, September 11, 2006 ; revised papers. ODBIS; International Workshop on Ontologies-based Techniques for DataBases and Information Systems; International VLDB Workshop on Ontologies-based Techniques for DataBases and Information Systems. Berlin: Springer (Lecture notes in computer science, 4623). Online verfügbar unter http://www.springerlink.com/openurl.asp?genre=issue&issn=0302-9743&volume=4623.

DEGLE, RALF (2007): Ontologie-basierte Beschreibung digitaler Lernmedien. 1. Aufl. München: GRIN.

DOR, DORIT (1995): Selection algorithms, zuletzt geprüft am 24.03.2016.

DOR, DORIT; ZWICK, URI (1999): Selecting the median. In: *SIAM Journal on Computing* 28 (5), S. 1722–1758.

EID, MOHAMAD; LISCANO, RAMIRO; EL SADDIK, ABDULMOTALEB (2007): A Universal Ontology for Sensor Networks Data. In: 2007 IEEE International Conference on Computational Intelligence for Measurement Systems and Applications. Ostuni, Italy, S. 59–62.

ELKAN, CHARLES; GREINER, RUSSELL (1993): Building large knowledge-based systems. Representation and inference in the cyc project. In: *Artificial Intelligence* 61 (1), S. 41–52. DOI: 10.1016/0004-3702(93)90092-P.

FENG, QIAN; LINWEN, XU (2008): Improving customer satisfaction by the expert system using artificial neural networks. In: 2008 7th World Congress on Intelligent Control and Automation. Chongqing, China, S. 8303–8306.

FERNANDES, LEANDRO A. F.; GARCÍA, ANA CRISTINA BICHARRA (2012): Association Rule Visualization and Pruning through Response-Style Data Organization and Clustering. In: David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern,

John C. Mitchell et al. (Hg.): Advances in Artificial Intelligence – IBERAMIA 2012, Bd. 7637. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture notes in computer science), S. 71–80.

FERNÁNDEZ-LÓPEZ, MARIANO; GÓMEZ-PÉREZ, ASUNCIÓN; Juristo Natalia (1997): METHONTOLOGY: From Ontological Art Towards Ontological Engineering (SS-97-06). In: *Proceedings of the Ontological Engineering AAAI-97 Spring Symposium Series*, S. 33–40.

FOREMAN, MATTHEW; KANAMORI, AKIHIRO (Hg.) (2010): Handbook of Set Theory. Dordrecht: Springer Science+Business Media B.V. Online verfügbar unter http://dx.doi.org/10.1007/978-1-4020-5764-9.

GOLUMBIC, MARTIN CHARLES (2004): Algorithmic graph theory and perfect graphs. Amsterdam, Boston: Elsevier (Annals of discrete mathematics, 57). Online verfügbar unter http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10191466.

GÓMEZ-PÉREZ, ASUNCIÓN; FERNÁNDEZ-LÓPEZ, MARIANO; CORCHO, OSCAR (2004): Ontological Engineering. With Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web. London: Springer-Verlag London Limited (Advanced Information and Knowledge Processing). Online verfügbar unter http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10130056.

GRUBER, THOMAS R. (1995): Toward principles for the design of ontologies used for knowledge sharing? In: *International Journal of Human-Computer Studies* 43 (5-6), S. 907–928. DOI: 10.1006/ijhc.1995.1081.

GRÜNINGER, MICHAEL; FOX, MARK S. (1995): Methodology for the Design and Evaluation of Ontologies.

GUAGLIANONE, M. T.; GUARASCI, R.; MATTA, N.; CAHIER, J.; BENEL, A. (Hg.) (2011): Comparison and evaluation of knowledge modeling techniques towards the definition of a global approach: MNEMO (Methodology for kNowledgE acquisition and MOdelling). Computer Sciences and Convergence Information Technology (ICCIT), 2011 6th International Conference on. Computer Sciences and Convergence Information Technology (ICCIT), 2011 6th International Conference on.

GUAN, JIAN; LEVITAN, ALAN S. (2012): A Model for Investigating Internal Control Weaknesses. In: *Communications of the Association for Information Systems* 31 (3).

GUARINO, NICOLA (1997): Semantic matching: Formal ontological distinctions for information organization, extraction, and integration. In: Jaime G. Carbonell, Jörg Siekmann, G. Goos, J. Hartmanis, J. Leeuwen und Maria Teresa Pazienza (Hg.): Information Extraction A Multidisciplinary Approach to an Emerging Information Technology, Bd. 1299. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture notes in computer science), S. 139–170.

GUIZZARDI, GIANCARLO; ZAMBORLINI, VERUSKA (2014): Using a trope-based foundational ontology for bridging different areas of concern in ontology-driven conceptual modeling. In: *Science of Computer Programming* 96, S. 417–443. DOI: 10.1016/j.scico.2014.02.022.

HAARMANN, BASTIAN (2014): Ontology On Demand. Vollautomatische Ontologieerstellung aus deutschen Texten mithilfe moderner Textmining-Prozesse. Univ., Fak. für Philologie, Diss.--Bochum, 2014. Berlin: epubli.

HAARSLEV, VOLKER (2016): Racer. Unter Mitarbeit von Ralf Möller und Michael Wessel. Universität zu Lübeck; Concordia University. Online verfügbar unter https://www.ifis.uni-luebeck.de/index.php?id=385, zuletzt geprüft am 17.03.2016.

HALL, JON G. (2012): Engineering knowledge engineering. In: *Expert Systems* 29 (5), S. 427–428. DOI: 10.1111/exsy.12007.

HARRISON, ROBERT; CHAN, CHRISTINE W. (2005): Implementation of an Application Ontology. In: Daoliang Li und Baoji Wang (Hg.): Artificial Intelligence Applications and Innovations, Bd. 187. New York: Springer-Verlag (IFIP — The International Federation for Information Processing), S. 131–143.

HASENKAMP, ULRICH; ROßBACH, PETER (1998): Wissensmanagement. In: *wisu* 27, 1998 (8-9), S. 956–964.

HEPP, MARTIN (2007): Possible Ontologies. How Reality Constrains the Development of Relevant Ontologies. In: *IEEE Internet Comput.* 11 (1), S. 90–96. DOI: 10.1109/MIC.2007.20.

HIRSCH, MANUEL (2011): Integration von Wissen in Innovationsnetzwerken mithilfe eines ontologiebasierten Entscheidungsunterstützungssystems. In: Meike Tilebein (Hg.): Innovation und Information. Wissenschaftliche Jahrestagung der Gesellschaft für Wirtschafts- und Sozialkybernetik vom 3. bis 5. Dezember 2008 in Oestrich-Winkel. Berlin: Duncker & Humblot (Wirtschaftskybernetik und Systemanalyse, 26), S. 295–309.

HOEHNDORF, ROBERT (2010): What is an upper level ontology? In: *Ontogenesis*. Online verfügbar unter http://ontogenesis.knowledgeblog.org/740.

HOPPE, UWE (1992): Methoden des Knowledge Engineering. Ein Expertensystem für das Wertpapiergeschäft in Banken. Wiesbaden: Deutscher Universitätsverlag. Online verfügbar unter http://dx.doi.org/10.1007/978-3-322-85952-5.

HORRIDGE, MATTHEW; KNUBLAUCH, HOLGER; RECTOR, ALAN; STEVENS, ROBERT; WROE, CHRIS (2004): A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools Edition 1.0. In: *University of Manchester*.

HWANG, GWO-JEN; CHEN, CHIEH-YUAN; TSAI, PEI-SHAN; TSAI, CHIN-CHUNG (2011): An expert system for improving web-based problem-solving ability of students. In: *Expert Systems with Applications* 38 (7), S. 8664–8672. DOI: 10.1016/j.eswa.2011.01.072.

JANSEN, LUDGER; SMITH, BARRY; BITTNER, THOMAS (Hg.) (2008): Biomedizinische Ontologie. Wissen strukturieren für den Informatik-Einsatz. Zürich: vdf-Hochschulverl.

KALANA MENDIS, D. S.; KARUNANANDA, ASOKA S.; SAMARATUNGA, U.; RATNAYAKE, U. (2007): Tacit knowledge modeling in Intelligent Hybrid systems. In: 2007 International Conference on Industrial and Information Systems. Peradeniya, Sri Lanka, S. 279–284.

KARAKOL, FATIH (2016): Konzeption einer ontologiebasierten Schnittstelle zur Integration von verteilt vorliegenden Informationsquellen: disserta Verlag. Online verfügbar unter https://books.google.de/books?id=f81wCwAAQBAJ.

KHELIF, KHALED; DIENG-KUNTZ, ROSE (2004): Ontology-Based Semantic Annotations for Biochip Domain. In: David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell et al. (Hg.): Engineering Knowledge in the Age of the Semantic Web, Bd. 3257. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture notes in computer science), S. 483–484.

KULINICH, A. A. (2012): Computer systems for cognitive maps simulation. Approaches and methods. In: *Autom Remote Control* 73 (9), S. 1553–1571. DOI: 10.1134/S0005117912090093.

LANDAUER, CHRISTOPHER (1998): Data, information, knowledge, understanding: computing up the meaning hierarchy. In: SMC '98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics. San Diego, CA, USA, 11-14 Oct. 1998, S. 2255–2260.

LINDBERG, D. A.; HUMPHREYS, B. L.; MCCRAY, A. T. (1993): The Unified Medical Language System. In: *Methods of information in medicine* 32 (4), S. 281–291.

LIU, XIJUAN; ROSEN, DAVID; YU, ZHONGHAI (2010): Ontology based knowledge modeling and reuse approach in product redesign. In: Integration (2010 IRI). Las Vegas, NV, USA, S. 270–273.

MASOLO, CLAUDIO; BORGO, STEFANO; GANGEMI, ALDO; GUARINO, NICOLA; OLTRAMARI, ALESSANDRO; SCHNEIDER, LUC (2003): WonderWeb Deliverable D17. The WonderWeb Library of Foundational Ontologies Preliminary Report. Padova, 29.05.2003. Online verfügbar unter http://www.loa.istc.cnr.it/old/Papers/DOLCE2.1-FOL.pdf.

MATTA, NADA; ERMINE, JEAN LOUIS; AUBERTIN, GÉRARD; TRIVIN, JEAN-YVES (2002): Knowledge Capitalization with a Knowledge Engineering Approach: The Mask Method. In: Rose Dieng-Kuntz und Nada Matta (Hg.): Knowledge Management and Organizational Memories. Boston, MA: Springer US, S. 17–28.

MCNEILL, FIONA; BUNDY, ALAN (2007): Dynamic, Automatic, First-Order Ontology repair by Diagnosis of Failed Plan Execution. In: *International Journal on Semantic Web and Information Systems* 3 (3), S. 1–35. DOI: 10.4018/jswis.2007070101.

MCNICHOLAS, PAUL D.; ZHAO, YANCHANG (2009): Association Rules. An Overview. In: Yanchang Zhao, Chengqi Zhang und Longbing Cao (Hg.): Post-mining of association rules. Techniques for effective knowledge extraction. Hershey, Pa.: Information Science Reference, S. 1–10.

MILLER, GEORGE A.; BECKWITH, RICHARD; FELLBAUM, CHRISTIANE; GROSS, DEREK; MILLER, KATHERINE J. (1990): Introduction to WordNet. An On-line Lexical Database *. In: *Int J Lexicography* 3 (4), S. 235–244. DOI: 10.1093/ijl/3.4.235.

MIZOGUCHI, RIICHIRO; KOZAKI, KOUJI; SANO, TOSHINOBU; KITAMURA, YOSHINOBU (2000): Construction and Deployment of a Plant Ontology. In: G. Goos, J. Hartmanis, J. van Leeuwen, Rose Dieng und Olivier Corby (Hg.): Knowledge Engineering and Knowledge Management Methods, Models, and Tools, Bd. 1937. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture notes in computer science), S. 113–128.

NGUYEN, DAT-DAO; KIRA, DENNIS S. (2000): Designing Optimal Knowledge Base for Integrated Neural Expert Systems. In: *AMCIS 2000 Proceedings*, S. 89.

NILES, IAN; PEASE, ADAM (2001): Towards a standard upper ontology. In: Nicola Guarino, Barry Smith und Christopher Welty (Hg.): the international conference. Ogunquit, Maine, USA, S. 2–9.

NORVIG, PETER (1992): Paradigms of Artificial Intelligence Programming. Case Studies in Common Lisp. 1. Aufl. s.l.: Elsevier Reference Monographs. Online verfügbar unter http://gbv.eblib.com/patron/FullRecord.aspx?p=1876698.

NOY, NATASHA; RECTOR, ALAN (2006): Defining N-ary Relations on the Semantic Web. Hg. v. w3c. Stanford University; University of Manchester. Online verfügbar unter https://www.w3.org/TR/swbp-n-aryRelations/, zuletzt geprüft am 11.03.2016.

ORTLEPP, DIANA (2016): Modellierung einer Wissensbasis mit iknow anhand von Scrum. Hauptseminar. Universität Ilmenau, Ilmenau. Ingeneurinformatik.

PALMA, JOSE; JUAREZ, JOSE M.; CAMPOS, MANUEL; MARIN, ROQUE (2006): Fuzzy theory approach for temporal model-based diagnosis: An application to medical domains. In: *Artificial intelligence in medicine* 38 (2), S. 197–218. DOI: 10.1016/j.artmed.2006.03.004.

PEPPER, STEVE (2000): The TAO of topic maps. Online verfügbar unter http://badame.vse.cz/2005/tao/TheNewTAO.pdf, zuletzt geprüft am 04.02.2016.

RAMIREZ, CARLOS; VALDES, BENJAMIN (2011): Memory Map of a Knowledge Representation model used for intelligent personalization of learning activities sequences. In: Cognitive Computing (ICCI-CC). Banff, AB, Canada, S. 423–431.

ROBINSON, EDWARD HEATH (2011): A Theory of Social Agentivity and its Integration into the Descriptive Ontology for Linguistic and Cognitive Engineering. In: *International Journal on Semantic Web and Information Systems* 7 (4), S. 62–86. DOI: 10.4018/ijswis.2011100103.

RUHE, GÜNTHER (2003): Software Engineering Decision Support – A New Paradigm for Learning Software Organizations. In: Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Scott Henninger und Frank Maurer (Hg.): Advances in Learning Software Organizations, Bd. 2640. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture notes in computer science), S. 104–113.

RUSSELL, STUART J.; NORVIG, PETER (2010): Artificial intelligence. A modern approach. 3. ed. Upper Saddle River, NJ: Prentice-Hall (Prentice-Hall series in artificial intelligence).

SAINT-AUBIN, YVAN; ROUSSEAU, CHRISTIANE (2008): Mathematics and Technology. New York, NY: Springer-Verlag New York (Springer Undergraduate Texts in Mathematics and Technology). Online verfügbar unter http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10257944.

SCHREIBER, GUUS; WIELINGA, BOB; JANSWEIJER, WOUTER (1995): The KACTUS View on the 'O' Word. Technical Report, ESPRIT Project 8145 KACTUS, University of Amsterdam, The Netherlands.

TIAN, TIAN; MINZHE, ZHU; BOMING, ZHANG (1995): An artificial neural network-based expert system for network topological error identification. In: ICNN'95 - International Conference on Neural Networks. Perth, WA, Australia, 27 Nov.-1 Dec. 1995, S. 882–886.

TRIEBEL, ANNE (2015): Entwicklung eines Kriterienkataloges zur Analyse von Softwarelösungen während der Softwareproduktentwicklung. Hauptseminar. Universität Ilmenau, Ilmenau. Technische Informatik und Ingenieurinformatik.

USCHOLD, MIKE; GRUNINGER, MICHAEL (1996): Ontologies. Principles, methods and applications. In: *Knowl. Eng. Rev.* 11 (02), S. 93. DOI: 10.1017/S0269888900007797.

w3c (2016): OWL Web Ontology Language. XML Presentation Syntax. w3c. Online verfügbar unter https://www.w3.org/TR/owl-xmlsyntax/, zuletzt geprüft am 03.03.2016.

w3schools (2016): XML RDF. Hg. v. w3schools. Online verfügbar unter http://www.w3schools.com/xml/xml_rdf.asp, zuletzt geprüft am 25.02.2016.

WADHWA, SHALINI (2005): Teaching and the learning vocabulary. 1st ed. New Delhi: Sarup & Sons (Sarup teaching learning series, 12).

WEBSTER, JANE; WATSON, RICHARD T. (2002): Analyzing the Past to Prepare for the Future: Writing a Literature Review. In: *MIS Quarterly* 26 (2), S. xiii–xxiii.

ZHU, HONG (2005): Software design methodology. Oxford, Boston: Elsevier Butterworth-Heinemann. Online verfügbar unter http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=166217.

ZIMMERMANN, ALBERT (1998): Ontologie oder Metaphysik? Die Diskussion über den Gegenstand der Metaphysik im 13. und 14. Jahrhundert ; Texte und Untersuchungen. Univ., Habil.-Schr./62--Köln, 1961. 2., erw. Aufl. Leuven: Peeters (Recherches de théologie et philosophie médiévales Bibliotheca, 1).

Anhang

Abstrakte Ebenen: Daten, Informationen und Wissen

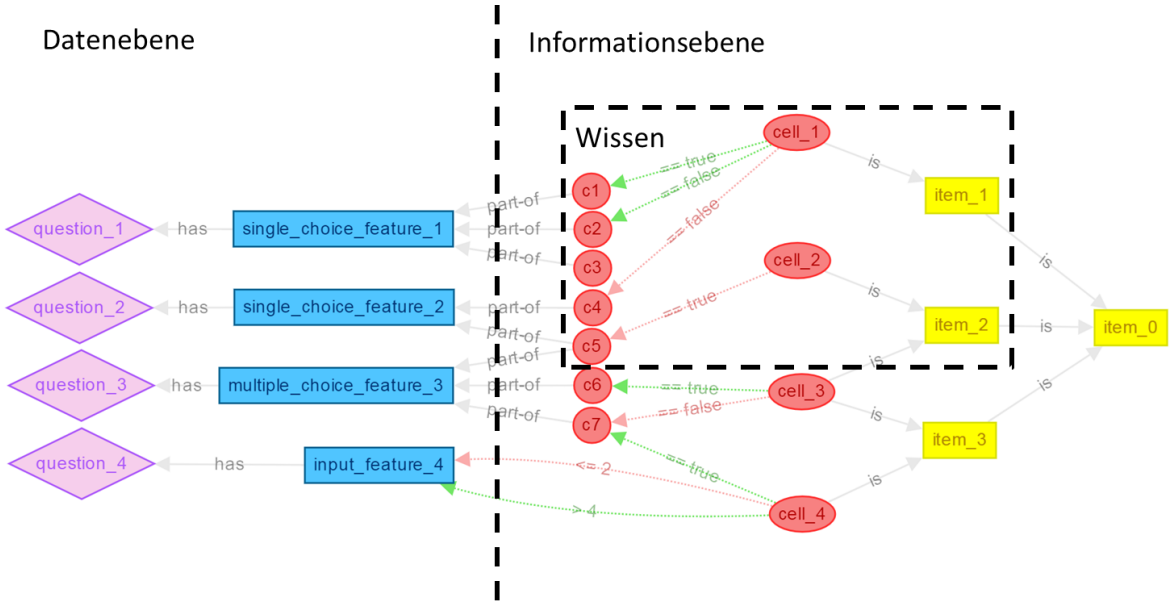


Abb. 46 Daten, Informationen und Wissen: Komponenten des Modells

Modellierung komplexer Wissenszusammenhänge am Beispiel „Scrum“

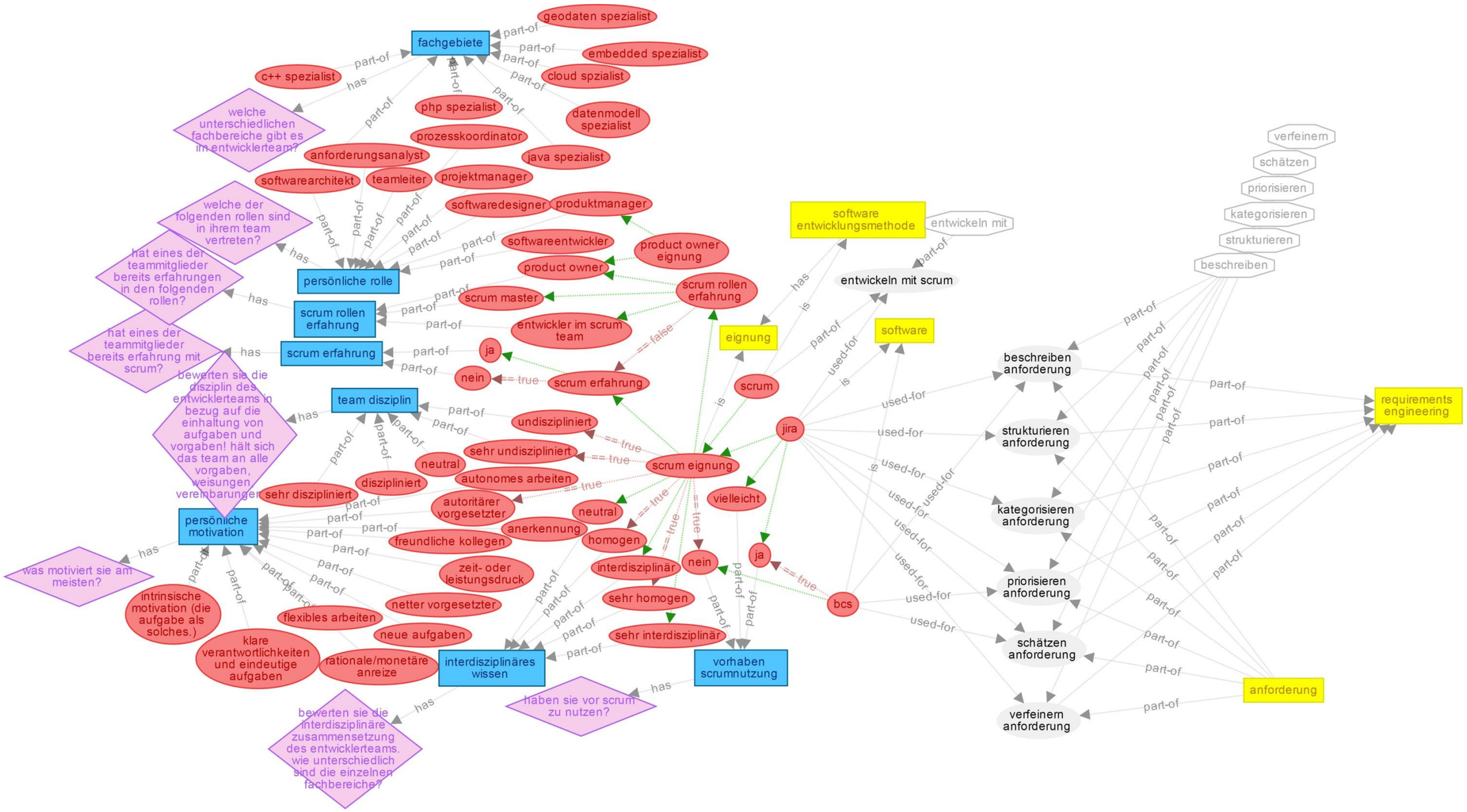


Abb. 47 Beispiele zur Modellierung komplexer Wissenszusammenhänge

Katalog zur Erfassung von Softwarevergleichskriterien

Tabelle 18 Katalog mit ausgewählten Kriterien für den Softwarevergleich

					Kriterien	JIRA	Eclipse	XStudio	MS Visio
A					**Anforderungsmanagement**	**76,19%**			
A	1				Anforderungsermittlung	20,00%			
A	1	1	1		Mindmapping	Astah			
A	1	1	2		Brainstorming	-			
A	1	1	3		Rollenspiele	-			
A	1	1	4		Interview	-			
A	1	1	5		Modellbasierte Szenarien	?			
A	2				Anforderungsdokumentation	85,71%			
A	2	1			Erfassen der Akzeptanzkriterien	x			
A	2	2			Funktionale Anforderungen	x			
A	2	3			Nichtfunktionale Anforderungen	x			
A	2	4			Erstellung von Wireframes	Gliffy			
A	2	5			Templateunterstützung (z.B. Volere)	x			
A	2	6			auto. Generierung Lasten-/Pflichtenheft	-			
A	2	7			Erstellen von Use-Case-Diagrammen	Plant UML			
A	3				Anforderungsverwaltung	100,00%			
A	3	1			Identifikation als Projektitem	x			
A	3	2			Allgemeine Lebensstati	x			
A	3	3			Strukturierung der Anforderungen	x			
A	3	4			Anfordungsschätzung	x			
A	3	5			Priorisierung der Anforderungen	x			
A	3	6			Kategorisierung der Anforderungen	x			
A	3	7			Ticketsystem	x			
A	3	8			Änderungshistorie	x			
A	3	9			Filtern von Anforderungen	x			
E					**Projektmanagement**	**95,45%**	**27,27%**	**40,91%**	**27,27%**
E	1				Benutzerverwaltung	100,00%	0,00%	100,00%	0,00%
E	1	1			Benutzerprofile	x	-	x	-
E	1	2			Benutzerrechte	x	-	x	-
E	1	3			Zugriffsrechte	x	-	x	-
E	2				Kommunikation	100,00%	0,00%	22,22%	33,33%
E	2	1			Nachrichten	HipChat	-	-	-
E	2	2			Chat	HipChat	-	-	-
E	2	3			Kommentare	x	-	-	-
E	2	4			Anhängen von Dateien	x	-	x	x
E	2	5			Reminder	x	-	-	-
E	2	6			Push-Nachrichten (eMail)	x	-	-	-
E	2	7	1		Export/Import CSV	x	-	x	-
E	2	7	2		Export/Import Word	x	-	-	x
E	2	7	3		Export/Import PDF	x	-	-	x

E	3			Projektstrukturplan	100,00%	20,00%	0,00%	0,00%
E	3	1		Ziele	x	-	-	-
E	3	2		Meilensteinplanung	x	-	-	-
E	3	3		Zeiten/Dauer	x	Rabbit	-	-
E	3	4		Ressourcenmanagement	x	-	-	-
E	3	5		Kosten-/Budgetmanagement	x	-	-	-
E	4			Projektkontrolle	x	x	x	-
E	5			Softwareentwicklungsmethode	50,00%	100,00%	100,00%	100,00%
E	5	1		Wasserfall	-	x	x	x
E	5	2		Scrum	x	x	x	x
E	6			Schnittstellenverfügbarkeiten	100,00%	100,00%	50,00%	50,00%
E	6	1		Schnittstelle zu ERP Systemen	x	x	?	x
E	6	2		Schnittstelle zu verwendetes Ticketsystem	x	x	x	?
X				**Systemvoraussetzungen**				
X	1			Arbeitsspeicher/RAM	2 GB	1 GB	1 GB	1 GB
X	2			Festplattenspeicherplatz	1 GB	1 GB	?	2 GB
X	3	1	1	CPU Intel	x	x	?	?
X	3	1	2	CPU AMD	?	x	?	?
X	3	1	3	CPU Sonstiger	-	?	?	-
X	3	2		CPU Plattform	32/64-bit	32/64-bit	?	32/64-bit
X	3	3		CPU GHz	2.27	1,5	1	1-2
X	4			bevorzugte Bildschrimauflösung	?	?	800×600	1024×576
X	5	1	1	GPU NVIDEA	?	?	?	?
X	5	1	2	GPU Intel	?	?	?	?
X	5	1	3	GPU ATI	?	?	?	?
X	5	2		GPU RAM	?	?	?	?
X	5	3		GPU GHz	?	?	?	?
X	6	1	1	Windows 7	x	x	x	x
X	6	1	2	Windows 8	x	x	x	x
X	6	2		MAC OS X 10.9	x	x	x	-
X	6	3		Linux Mint 17	x	x	x	-
X	6	4		Sonstiges	-	?	-	-
X	7	1		Webbasiert	x	-	-	-
X	7	2		Anwendungssoftware lokal	x	x	x	x
X	7	3		Cloudbasiert	-	-	x	-
Y				**Kosten**				
Y	1			Preis	ab 10$/M	kostenfrei	ab 179$/J	399 €
Y	2			Einmalzahlung	x	-	-	x
Y	3			Abonnement	x	-	x	-
Y	4			Open Source	x	-	-	-

Überführung des Softwarevergleichs in das Modell

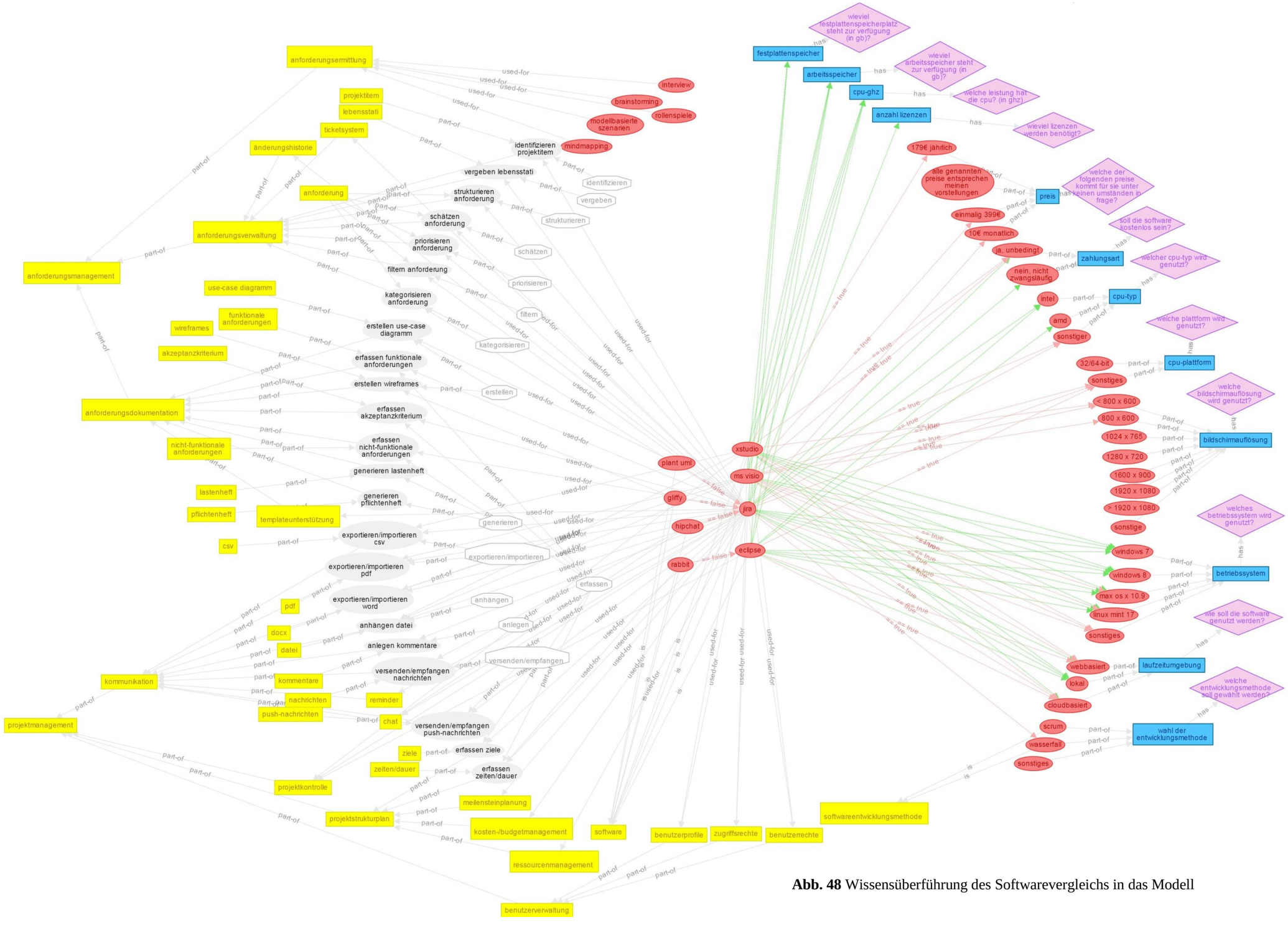


Abb. 48 Wissensüberführung des Softwarevergleichs in das Modell

Testfälle zur Überprüfung der Anforderungen an die GUI

Tabelle 19 Testfälle zur Überprüfung der GUI

Id	A.-Id	Testfall
T01	A01	Nutzer legt Cell an. Er gibt den Namen der Cell aus beliebigen alphanumerischen Zeichen ein.
		Erwartetes Ergebnis: Rote Ellipse erscheint auf der Zeichenfläche. Darin steht der Name der Cell.
T02	A01	Wie T01, nur mit einem Item.
		Erwartetes Ergebnis: Gelbes Rechteck erscheint auf der Zeichenfläche. Darin steht der Name des Items.
T03	A01	Wie T01, nur mit einem Feature. Nutzer wählt über ein Auswahlfeld den Featuretyp "Input Feature" aus.
		Erwartetes Ergebnis: Blaues Rechtecke erscheint auf der Zeichenfläche. Darin steht der Name des Feature.
T04	A01	Wie T01, nur mit einer Activity
		Erwartetes Ergebnis: Weißes Sechseck erscheint auf der Zeichenfläche. Darin steht der Name der Activity.
T05	A01	Wie T01, nur mit einem Combining. Nutzer gibt als Activity das Element aus T04 an und als Objekt das Element aus T01.
		Erwartetes Ergebnis: Graue Ellipse erscheint auf der Zeichenfläche. Darin stehen die Namen der kombinierten Elemente (mit einem Leerzeichen getrennt). Zusätzlich sind part-of-Verbindungen zwischen den Elementen aus T04 und dem neuen Combining und T01 und dem neuen Combining vorhanden.
T06	A01	Wie T01, nur mit einer Question. Als Namen gibt der Nutzer eine Fragestellung ein.
		Erwartetes Ergebnis: Lilafarbene Raute erscheint auf der Zeichenfläche. Darin steht die Fragestellung der Question.
T07	A02	Nutzer legt eine is-Assoziation zwischen der Cell (Source) aus T01 und dem Item (Destination) aus T02 an.

		Erwartetes Ergebnis: Eine durchgezogene Verbindung zwischen den Elementen erscheint auf der Zeichenfläche. Sie enthält die Bezeichnung "is".
T08	A02	Nutzer legt eine used-for-Assoziation mit Source T01 und Destination T04 an.
		Erwartetes Ergebnis: Eine used-for-Verknüpfung wird zwischen der gewählten Source und Destination erzeugt.
T09	A02	Nutzer erstellt neue Cell und neue Activity. Anschließend legt er eine can-Assoziation an. Dabei wählt er als die neue Activity und als Zielknoten die neue Cell. Source bildet die Cell aus T01.
		Erwartetes Ergebnis: Die neuen Activity und Cell erscheinen auf der Zeichenfläche. Zwischen der Cell aus T01 und der neuen Activity existiert eine can-Verbindung. Zusätzlich erscheint ein Combining aus den neuen Elementen und der entsprechenden part-of-Verbindungen.
T10	A02	Nutzer erstellt eine Negative-Constraint von Cell aus T01 und Cell aus T08. Als booleschen Ausdruck wird "==true" angegeben.
		Erwartetes Ergebnis: Zwischen den Cells existiert eine rote, gepunktete Linie, auf der der "==true" steht.
T11	A03	Nutzer verschiebt ein beliebiges Element.
		Erwartetes Ergebnis: Element lässt sich per Drag'n'Drop verschieben.
T12	A03	Nutzer löscht ein selektiertes Element.
		Erwartetes Ergebnis: Element verschwindet von der Zeichenfläche.
T13	A03	Nutzer betätigt Taste "n" auf seiner Tastatur.
		Erwartetes Ergebnis: Es öffnet sich der Element anlegen-Dialog.
T14	A03	Nutzer betätigt Taste "e" auf seiner Tastatur.
		Erwartetes Ergebnis: Es öffnet sich der Assoziation erzeugen-Dialog.
T15	A03	Nutzer klickt auf Button Vollbild.
		Erwartetes Ergebnis: Die Zeichenfläche wird auf Vollbild im Browser dargestellt.
T16	A03	Nutzer betätigt Mausrad.
		Erwartetes Ergebnis: Zeichenfläche vergrößert bzw. verkleinert den Bildausschnitt.

T17	A04	Nutzer klickt auf Speichern-Button, gibt einen Namen für die Datei ein und Bestätigt den Dialog.
		Erwartetes Ergebnis: Es öffnet sich ein Speichern-Dialog mit einem Textfeld und einem Bestätigen Button. Nach Klicken auf Bestätigen wird die Datei heruntergeladen.
T18	A04	Nutzer klickt auf Öffnen-Button, wählt die zuvor heruntergeladene Datei aus und Bestätigt.
		Erwartetes Ergebnis: Alle zuvor angelegten Elemente existieren und befinden sich an den gleichen Positionen wie vor dem Speichern.
T19	A05	Nutzer selektiert die Cell aus T01 und klickt auf Bearbeiten. Im Dialog ändert er den Namen und klickt auf Speichern.
		Erwartetes Ergebnis: Der Bearbeiten-Dialog öffnet sich. Er enthält die ID des selektierten Elementes. Im Textfeld steht der aktuelle Name der Cell. Nach Klick auf "Bestätigen" erscheint der neue Name in der Cell.
T20	A06	Nutzer selektiert die Cell aus T01 und klickt auf Bearbeiten. Im Dialog fügt er einen booleschen Ausdruck in das Feld "Ergebnis erzeugen" ein. Dabei integriert er durch "r(feature_id_aus_T03)" das Ergebnis des Features aus T03 in den booleschen Ausdruck (die id des Features ist entsprechend zu ersetzen). Beispielsweise könnte der Ausdruck lauten "r(feautre_id_aus_T03) > 3". Anschließend bestätigt er die Eingabe im Dialogfenster.
		Erwartetes Ergebnis: Eine grün-gepunktete Linie wurde zwischen den Elementen Cell aus T01 und Feature aus T03 erzeugt.
T21	A06	Wie T20 nur mit anderen Operatoren im booleschen Ausdruck. Dazu ändert der Nutzer den vorhandenen booleschen Ausdruck der Cell aus T01.
		Erwartetes Ergebnis: Wird weiterhin nur auf das Feature aus T03 verwiesen, bleibt die GUI unverändert. Die Änderung kann überprüft werden, in dem der Bearbeiten-Button für diese Cell erneut geklickt wird.
T22	A07	Nutzer lädt die von ihm erstellte Ontologie mit dem booleschen Ausdruck aus T20. Anschließend klickt er auf den Button "Isn't it?". Im Dialog wählt er die Cell aus T01 durch Eingabe des entsprechenden Namens und bestätigt den Dialog.

		Erwartetes Ergebnis: Ein Dialog öffnet sich mit einem Textfeld zur Eingabe eines Elementnamens. Nach Eingabe eines Buchstabens öffnet sich ein Autocomplete-Feld zur Auswahl eines Elementes. Nach Bestätigen des Dialogs öffnet sich ein neues Dialogfenster zur Eingabe des Feature-Wertes aus T03. Darin enthalten ist ein Textfeld und die Fragestellung der Question aus T06.
T22a	A07	Der Dialog zur Erfassung des Input-Wertes vom Feature aus T03 ist geöffnet. Nutzer gibt einen Wert ein, der zu dem booleschen Ausdruck aus T20 passt und bestätigt den Dialog.
		Erwartetes Ergebnis: Der Feature-Dialog schließt sich. Auf der Browserseite erscheint ein Text mit dem entsprechenden Ergebnis.
T23	A07	Nutzer lädt eine Ontologie mit mehreren unterschiedlichen Featuretypen und Cells, die teilweise einem single-choice- oder multiple-choice-Feature per part-of-Verknüpfung zugeordnet sind. Anschließend klickt er auf den "Isn't It?"-Button und wählt eine Cell, deren Ergebnis gefunden werden soll.
		Erwartetes Ergebnis: Alle Feature-Dialoge enthalten die entsprechenden Formularfelder. Die Auswahlmöglichkeiten bei single-choice- und multiple-choice-Featuren entsprechen den verknüpften Cells.
T24	A07	Nutzer lädt die Ontologie aus T17 und klickt anschließend auf den "Find!"-Button.
		Erwartetes Ergebnis: Ein Dialog mit zwei Textfelder zur Auswahl der Klasse und der Aktivität öffnet sich.
T24a	A07	Nutzer gibt in das Klassen-Textfeld aus T24 das Item aus T02 und im Aktivität-Textfeld die Activity aus T04 ein. Anschließend bestätigt er den Dialog.
		Erwartetes Ergebnis: Der Name der Cell aus T01 erscheint im Browserfenster als Ergebnis des Find!-Algorithmus.

Testfälle zur Überprüfung ausgewählter Quellcode-Elemente

Tabelle 20 Testfälle zur Überprüfung des Quellcodes

Id	Klasse	Testfall
T25	Graph	$graph = new Graph("test_ontology_1.json"); $a = $graph->getNodeById("C1");
		Erwartetes Ergebnis: $a ist ein Objekt vom Typ Cell mit der Id "C1", dem Namen "testnode", dem Status "created", keinem Ergebnis und keinem boole-schen Ausdruck.
T26	Graph	$graph = new Graph("test_ontology_1.json"); $a = $graph->getNodeByName("testnodeC2");
		Erwartetes Ergebnis: $a ist ein Objekt vom Typ Cell mit der Id "C2", dem Namen "testnode2", dem Status "created", keinem Ergebnis und einer expressi-on "r(C1)".
T27	Graph	$graph = new Graph("test_ontology_1.json"); $a = $graph->getNodeById("F1");
		Erwartetes Ergebnis: $a ist ein Objekt vom Typ Feature mit der Id "F1", dem Namen testfeature, dem Status "created", ohne Ergebnis, ohne expression und dem featureType "input"
T28	Graph	$graph = new Graph("test_ontology_1.json"); $graph->saveStm(null, "test_ontology_1a");
		Erwartetes Ergebnis: Ordner data/stmemory enthält eine Datei „test_ontology_1a.json“ mit den Knoten aus test_ontology_1.json
T29	Graph	$graph = new Graph("test_ontology_1.json"); $a = $graph->getNodes();
		Erwartetes Ergebnis: $a ist ein Array, das alle Knoten aus test_ontology_1.json enthält.
T30	Node	$graph = new Graph("test_ontology_2.json"); $a = $graph->getNodeById("I1"); $pre = $a->getPredecessors();
		Erwartetes Ergebnis: $pre ist ein Array, das alle Vorgängerknoten von I1 enthält: C1, C2, C3
T31	Node	$graph = new Graph("test_ontology_2.json"); $a = $graph->getNodeById("C1"); $suc = $a->getSuccessors();

		Erwartetes Ergebnis: $suc ist ein Array, das alle Nachfolgerknoten von C1 enthält: I1, I2
T32	Node	$graph = new Graph("test_ontology_2.json"); $a = $graph->getNodeById("I1"); $b = $a->getIsntItNodes();
		Erwartetes Ergebnis: $b ist ein Array, das alle relevanten Knoten zur Durchführung des Isn't It?-Algorithmus von I1 enthält: I1, C1, C2, C3, F1, F2, Q1
T33	Node	$graph = new Graph("test_ontology_2.json"); $a = $graph->getNodeById("A1"); $b = $a->findUsedForNodes();
		Erwartetes Ergebnis: $b ist ein Array, das alle relevanten used-for-Knoten von A1 enthält: C3
T34	Node	$graph = new Graph("test_ontology_2.json"); $a = $graph->getNodeById("I1"); $b = $a->findUsedForNodes();
		Erwartetes Ergebnis: $b ist ein Array, das alle relevanten used-for-Knoten von I1 enthält: C3
T35	Node	$graph = new Graph("test_ontology_2.json"); $a = $graph->getNodeById("F1"); $a->setResult("5");
		Erwartetes Ergebnis: Das Ergebnis von $a ist 5. Alle anderen Knoten des Graphs besitzen noch kein Ergebnis.
T36	Node	$graph = new Graph("test_ontology_2.json"); $a = $graph->getNodeById("F1"); $b = $graph->getNodeById("F2"); $a->setResult("5"); $b->setResult("9");
		Erwartetes Ergebnis: Das Ergebnis von $a ist 5. Das Ergebnis von $b ist 9. Das Ergebnis von C1 ist false. Das Ergebnis von C2 ist true. Das Ergebnis von C3 ist false. Das Ergebnis von I2 ist false. Das Ergebnis von I1 ist true.

Testdaten für Testfälle


```
[{
      "data": {
         "id": "C1",
         "name": "testnode",
         "type": "cell",
         "expression": "",
         "result": null,
         "featureType": null
      },
      "position": {
         "x": 0,
         "y": 0
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "classes": "cell"
   }, {
      "data": {
         "id": "C2",
         "name": "testnode2",
         "type": "cell",
         "expression": "",
         "result": null,
         "featureType": null
      },
      "position": {
         "x": 90,
         "y": 2
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "classes": "cell"
   },
```


Abb. 49 Testdaten in test_ontology_1.json, Auszug 1 von 2

```
{
      "data": {
         "id": "F1",
         "name": "testfeature",
         "type": "feature",
         "expression": null,
         "result": null,
         "featureType": "input"
      },
      "position": {
         "x": 40,
         "y": 67
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "classes": "feature"
}]
```


Abb. 50 Testdaten in test_ontology_1.json, Auszug 2 von 2


```
[{
      "data": {
         "id": "I1",
         "name": "i1",
         "type": "item",
         "expression": null,
         "result": null,
         "featureType": null
      },
      "group": "nodes",
      "classes": "item"
   },
```


Abb. 51 Testdaten in test_ontology_2.json, Auszug 1 von 6

```
{
   "data": {
      "id": "F1",
      "name": "c1",
      "type": "cell",
      "expression": "r(F1) > 5",
      "result": null,
      "featureType": null
   },
   "group": "nodes",
   "classes": "cell"
}, {
   "data": {
      "id": "F2",
      "name": "c2",
      "type": "cell",
      "expression": "r(F2) < 10",
      "result": null,
      "featureType": null
   },
   "group": "nodes",
   "classes": "cell"
}, {
   "data": {
      "id": "F3",
      "name": "c3",
      "type": "cell",
      "expression": "r(F2) > 10",
      "result": null,
      "featureType": null
   },
   "group": "nodes",
   "classes": "cell"
},{
   "data": {
      "id": "E1",
      "source": "C1",
      "target": "I1",
      "type": "is",
      "label": "is"
   },
   "group": "edges",
   "classes": "required"
},
```


Abb. 52 Testdaten in test_ontology_2.json, Auszug 2 von 6

```
  {
    "data": {
      "id": "E2",
      "source": "C2",
      "target": "I2",
      "type": "is",
      "label": "is"
    },
    "group": "edges",
    "classes": "required"
  }, {
    "data": {
      "id": "E3",
      "source": "C3",
      "target": "I2",
      "type": "is",
      "label": "is"
    },
    "group": "edges",
    "classes": "required"
  }, {
    "data": {
      "id": "I2",
      "name": "i2",
      "type": "item",
      "expression": null,
      "result": null,
      "featureType": null
    },
    "group": "nodes",
    "classes": "item"
  }, {
    "data": {
      "id": "E4",
      "source": "C1",
      "target": "I2",
      "type": "is",
      "label": "is"
    },
    "group": "edges",
    "classes": "required"
  },
```


Abb. 53 Testdaten in test_ontology_2.json, Auszug 3 von 6

```
{
   "data": {
      "id": "F1",
      "name": "f1",
      "type": "feature",
      "expression": null,
      "result": null,
      "featureType": "input"
   },
   "group": "nodes",
   "classes": "feature"
}, {
   "data": {
      "id": "F2",
      "name": "f2",
      "type": "feature",
      "expression": null,
      "result": null,
      "featureType": "input"
   },
   "group": "nodes",
   "classes": "feature"
}, {
   "data": {
      "id": "E5",
      "source": "C1",
      "target": "F1",
      "type": "positive-constraint",
      "label": ""
   },
   "group": "edges",
   "classes": "positive-constraint"
}, {
   "data": {
      "id": "E6",
      "source": "C2",
      "target": "F2",
      "type": "positive-constraint",
      "label": ""
   },
   "group": "edges",
   "classes": "positive-constraint"
},
```


Abb. 54 Testdaten in test_ontology_2.json, Auszug 4 von 6

```
  {
    "data": {
      "id": "E7",
      "source": "C3",
      "target": "F2",
      "type": "positive-constraint",
      "label": ""
    },
    "group": "edges",
    "classes": "positive-constraint"
  }, {
    "data": {
      "id": "A1",
      "name": "a1",
      "type": "activity",
      "expression": null,
      "result": null,
      "featureType": null
    },
    "group": "nodes",
    "classes": "activity"
  }, {
    "data": {
      "id": "I3",
      "name": "i3",
      "type": "item",
      "expression": "",
      "result": null,
      "featureType": null
    },
    "group": "nodes",
    "classes": "item"
  }, {
    "data": {
      "id": "E8",
      "source": "C3",
      "target": "A1",
      "type": "used-for",
      "label": "used-for"
    },
    "group": "edges",
    "classes": "required"
  },
```


Abb. 55 Testdaten in test_ontology_2.json, Auszug 5 von 6

```
  {
    "data": {
      "id": "E9",
      "source": "A1",
      "target": "I3",
      "type": "part-of",
      "label": "part-of"
    },
    "group": "edges",
    "classes": "required"
  }, {
    "data": {
      "id": "Q1",
      "name": "enter value",
      "type": "question",
      "expression": null,
      "result": null,
      "featureType": null
    },
    "group": "nodes",
    "classes": "question"
  }, {
    "data": {
      "id": "E10",
      "source": "F1",
      "target": "Q1",
      "type": "has",
      "label": "has"
    },
    "group": "edges",
    "classes": "required"
  }, {
    "data": {
      "id": "E11",
      "source": "F2",
      "target": "Q1",
      "type": "has",
      "label": "has"
    },
    "group": "edges",
    "classes": "required"
}]
```

Abb. 56 Testdaten in test_ontology_2.json, Auszug 6 von 6

Ergebnisse der Literaturrecherche

Tabelle 21 Ergebnisse der Literaturrecherche

DOI	Titel	Autor(en)	publiziert in
10.1016/j.eswa.2013.05.040	JOINT: Java ontology integrated toolkit	Olavo Holandaa, Seiji Isotanib, Ig Ibert Bittencourta, Endhe Eliasa, Thyago Tenórioa	Expert Systems with Applications Volume 40, Issue 16, 15 November 2013, Pages 6469–6477
doi:10.1016/j.datak.2013.06.004	An autonomic ontology-based approach to manage information in home-based scenarios: From theory to practice	N. Lasierraa, A. Alesancoa, D. O'Sullivanb, J. Garcíaa	Data & Knowledge Engineering Volume 87, September 2013, Pages 185–205
10.1007/978-3-642-02121-3_20	A Core Ontology of Knowledge Acquisition	José Iria	The Semantic Web: Research and Applications Volume 5554 of the series Lecture Notes in Computer Science pp 233-247
10.1016/j.eswa.2010.05.009	Supporting small teams in cooperatively building application domain models	Cesar Augusto Taclaa, Ademir Roberto Freddoa, Emerson Cabrera Paraisob, Milton Pires Ramosc, Gilson Yukio Satoa	Expert Systems with Applications Volume 38, Issue 2, February 2011, Pages 1160–1170
doi:10.1016/j.is.2014.05.002	Identification of ontologies to support information systems development	Ghassan Beydouna, Graham Lowb, Francisco García-Sánchezc, Rafael Valencia-Garcíac, Rodrigo Martínez-Béjarc	Information Systems Volume 46, December 2014, Pages 45–60
10.1016/j.ijinfomgt.2014.06.005	Ontology for knowledge management in software maintenance	Edgar Serna M.a, Alexei Serna A.b	International Journal of Information Management Volume 34, Issue 5, October 2014, Pages 704–710

10.1504/IJMSO.2011.046595	Formal modelling, knowledge representation and reasoning for design and development of user-centric pervasive software: a meta-review	Ahmet Soylu, Patrick De Causmaecker, Davy Preuveneers, Yolande Berbers, Piet Desmet	International Journal of Metadata, Semantics and Ontologies
10.1016/j.accinf.2012.08.002	How AIS can progress along with ontology research in IS	Jian Guana, Alan S. Levitanb, John R. Kuhn Jr.c,	International Journal of Accounting Information Systems Volume 14, Issue 1, March 2013, Pages 21–38
10.1007/978-3-642-16438-5_25	A Model-Driven Approach for Using Templates in OWL Ontologies	Fernando Silva Parreiras, Gerd Gröner, Tobias Walter, Steffen Staab	Knowledge Engineering and Management by the Masses Volume 6317 of the series Lecture Notes in Computer Science pp 350-359
10.1109/SBES.2014.13	A Systematic Review on the Use of Ontologies in Requirements Engineering	Diego Dermeval, Jessyka Vilela , Ig Ibert Bittencourt, Jaelson Castro, Seiji Isotani, Patrick Brito	Software Engineering (SBES), 2014 Brazilian Symposium on Sept. 28 2014-Oct. 3 2014
10.1016/j.csi.2010.12.001	A framework for reviewing domain specific conceptual models	Ö. Özgür Tanrıövera, Semih Bilgen	Computer Standards & Interfaces Volume 33, Issue 5, September 2011, Pages 448–464
10.1109/ICTKE.2012.6152410	Reusing OWL-S to model knowledge intensive tasks performed by Knowledge Based Systems	Baby A. Gobin	2011 Ninth International Conference on ICT and Knowledge Engineering 12-13 Jan. 2012, Pages 34-42
10.1504/IJMSO.2010.033283	A systematic review of research on integration of ontologies with the model-driven approach	Maria-Cruz Valiente	International Journal of Metadata, Semantics and Ontologies
10.17485/ijst/2016/v9i9/71384	Ontologies for Software Engineering: Past, Present and Future	M. P. S. Bhatia, Akshi Kumar, Rohit Beniwal	Indian Journal of Science and Technology, Vol 9(9), DOI: 10.17485/ijst/2016/v9i9/71384, March 2016

10.1109/SEAA.2014.78	A Proposal of an Ontology-Based System for Distributed Teams	Rodrigo G. C. Rocha, Ryan Azevedo, Silvio Meira	2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications 27-29 Aug. 2014, Pages 398-401
10.1016/j.scico.2014.02.022	Using a trope-based foundational ontology for bridging different areas of concern in ontology-driven conceptual modeling	Giancarlo Guizzardia, Veruska Zamborlini	Science of Computer Programming Volume 96, Part 4, 15 December 2014, Pages 417–443
10.1016/j.eswa.2013.05.007	Formal Concept Analysis in knowledge processing: A survey on models and techniques	Jonas Poelmansa, Sergei O. Kuznetsovb, Dmitry I. Ignatovb, Guido Dedene	Expert Systems with Applications Volume 40, Issue 16, 15 November 2013, Pages 6601–6623
10.1016/j.datak.2010.01.002	Towards automatization of domain modeling	Iris Reinhartz-Berger	Data & Knowledge Engineering Volume 69, Issue 5, May 2010, Pages 491–515
10.1007/978-3-540-72586-2_162	Epistemological and Ontological Representation in Software Engineering	J. Cuadrado-Gallego, D. Rodríguez, M. Garre, R. Rejas	Computational Science – ICCS 2007 Volume 4488 of the series Lecture Notes in Computer Science pp 1162-1169
10.1007/978-3-319-23781-7_1	Semi-automated Generation of DSL Meta Models from Formal Domain Ontologies	Andres Ojamaa, Hele-Mai Haav , Jaan Penjam	Model and Data Engineering Volume 9344 of the series Lecture Notes in Computer Science pp 3-15
10.1002/spe.1132	Ontology patterns for service-oriented software development	Wei-Tek Tsai, Budan Wu, Zhi Jin, Yu Huang, Wu Li	Software: Practice and Experience Special Issue: Pattern Languages: Addressing the Challenges Volume 43, Issue 7, pages 867–883, July 2013
	Ontology Modeling and Object Modeling in Software Engineering	Waralak V. Siricharoen	International Journal of Software Engineering and Its Applications Vol. 3, No. 1, January, 2009

10.1109/CSA.2008.9	A Software Engineering Approach to Comparing Ontology Modeling with Object Modeling	Waralak V. Siricharoen	Computer Science and its Applications, 2008. CSA'08. International Symposium on. IEEE, 2008.
10.1145/568760.568822	An ontological approach to domain engineering	Ricardo de Almeida Falbo, Giancarlo Guizzardi, Katia Cristina Duarte	SEKE '02 Proceedings of the 14th international conference on Software engineering and knowledge engineering Pages 351-358
10.1145/1458484.1458487	An interactive ontology learning workbench for non-experts	Jon Atle Gulla, Vijayan Sugumaran	ONISW '08 Proceedings of the 2nd international workshop on Ontologies and information systems for the semantic web Pages 9-16
10.1007/11961239_2	Ad-Hoc and Personal Ontologies: A Prototyping Approach to Ontology Engineering	Debbie Richards	Advances in Knowledge Acquisition and Management Volume 4303 of the series Lecture Notes in Computer Science pp 13-24
10.1080/00207543.2014.919422	An ontology framework for developing platform-independent knowledge-based engineering systems in the aerospace industry	I.O. Sanya, E.M. Shehab	International Journal of Production Research Volume 52, Issue 20, 2014, pages 6192-6215
10.1016/j.knosys.2007.02.001	Concept similarity in Formal Concept Analysis: An information content approach	Anna Formica	Knowledge-Based Systems Volume 21, Issue 1, February 2008, Pages 80–87
10.1142/S0218194010004876	DEFINING SOFTWARE PROCESS MODEL CONSTRAINTS WITH RULES USING OWL AND SWRL	DANIEL RODRÍGUEZ, ELENA GARCÍA, SALVADOR SÁNCHEZ, CARLOS RODRÍGUEZ-SOLANO NUZZI	International Journal of Software Engineering and Knowledge Engineering, June 2010, Vol. 20, No. 04 : pp. 533-548
10.1016/j.cad.2012.08.006	The evolution, challenges, and future of knowledge representation in product design systems	Senthil K. Chandrasegarana, Karthik Ramania, Ram D. Sriramc, Imré Horváthd, Alain Bernarde, Ramy F. Harikf, Wei Gaoa	Computer-Aided Design Volume 45, Issue 2, February 2013, Pages 204–228 Solid and Physical Modeling 2012

10.1145/2522968.2522971	Organizational social structures for software engineering	Damian A. Tamburri, Patricia Lago, Hans van Vliet	ACM Computing Surveys (CSUR) Surveys Homepage archive Volume 46 Issue 1, October 2013 Article No. 3
10.1049/iet-sen:20070062	Ontological approach for the semantic recovery of traceability links between software artefacts	Y. Zhan, R. Witte, J. Rilling, V. Haarslev	IET Software (Volume:2 , Issue: 3) pages 185 - 203
10.1017/S0269888899900087	Ontologies of engineering knowledge: general structure and the case of Software Engineering	Miguel-Angel Sicilia, Elena García-Barriocanal, Salvador Sánchez-Alonso, Daniel Rodríguez-García	The Knowledge Engineering Review The Knowledge Engineering Review / Volume 24 / Special Issue 03 / September 2009, pp 309-326
10.1016/j.datak.2010.10.001	Supporting concurrent ontology development: Framework, algorithms and tool	E. Jiménez Ruiz, B. Cuenca Grau, I. Horrocks, R. Berlang	Data & Knowledge Engineering Volume 70, Issue 1, January 2011, Pages 146–164
10.1109/CISIS.2009.90	Making Expert Knowledge Explicit to Facilitate Tool Support for Integrating Complex Information Systems in the ATM Domain	Thomas Moser, Richard Mordinyi, Alexander Mikula, Stefan Biffl	Complex, Intelligent and Software Intensive Systems, 2009. CISIS '09. International Conference on pages 90 - 97
10.1007/s11042-012-1134-9	Semantics enhanced engineering and model reasoning for control application development	David Hästbacka , Seppo Kuikka	Multimedia Tools and Applications July 2013, Volume 65, Issue 1, pp 47-62
10.1016/j.compind.2014.04.006	An exploratory study on ontology engineering for software architecture documentation	K.A. de Graaf, P. Liang, A. Tang, W.R. van Hage, H. van Vliet	Computers in Industry Volume 65, Issue 7, September 2014, Pages 1053–1064
10.1016/j.is.2008.07.002	A software engineering approach to ontology building	Antonio De Nicola, Michele Missikoff, Roberto Navigli	Information Systems Volume 34, Issue 2, April 2009, Pages 258–275

10.1016/j.jss.2010.10.025	Bridging metamodels and ontologies in software engineering	B. Henderson-Sellers	Journal of Systems and Software Volume 84, Issue 2, February 2011, Pages 301–313
10.1016/j.eswa.2012.04.009	Towards an ontology modeling tool. A validation in software engineering scenarios	Francisco José García-Peñalvo, Ricardo Colomo-Palacios, Juan García, Roberto Therón	Expert Systems with Applications Volume 39, Issue 13, 1 October 2012, Pages 11468–11478
10.1142/S0218194004001646	AN ONTOLOGY FOR THE MANAGEMENT OF SOFTWARE MAINTENANCE PROJECTS	FRANCISCO RUIZ, AURORA VIZCAÍNO, MARIO PIATTINI, FELIX GARCÍA	International Journal of Software Engineering and Knowledge Engineering Volume 14, Issue 03, June 2004
10.1007/978-3-540-40052-3_10	Software Engineering Decision Support – A New Paradigm for Learning Software Organizations	Günther Ruhe	Advances in Learning Software Organizations Volume 2640 of the series Lecture Notes in Computer Science pp 104-113
	Requirements for a Tool in Support of SE Technology Selection	Jedlitschka, Andreas; Pfahl, Dietmar	16th International Conference on Software Engineering and Knowledge Engineering. SEKE'2004. Skokie : Knowledge Systems Institute, 2004, 513-516
	A Framework for Comprehensive Experience-based Decision Support for Software Engineering Technology Selection	Jedlitschka, Andreas; Pfahl, Dietmar; Bomarius, Frank	6th International Conference on Software Engineering and Knowledge Engineering. SEKE'2004. Skokie : Knowledge Systems Institute, 2004, pp. 342-345